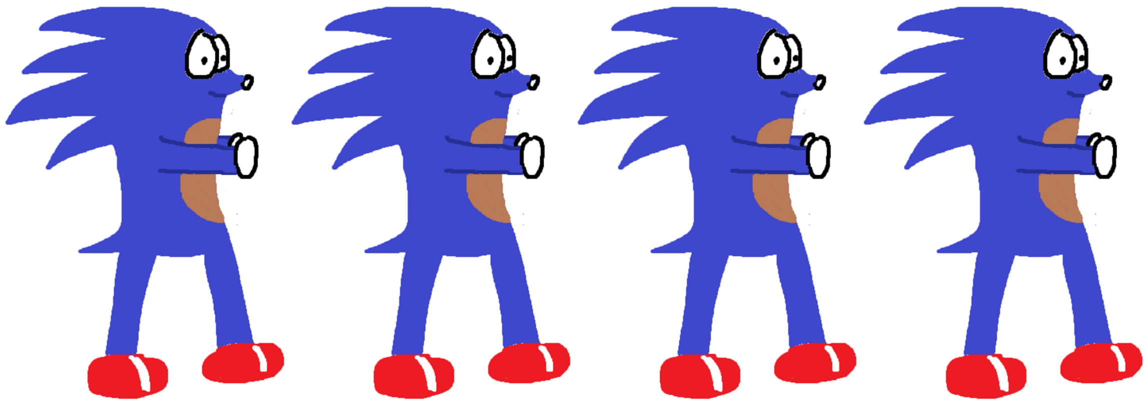
UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE



Bachelor project

W. Pema N. H. Malling <vkc347>
Louis Marott Normann <cqb868>
Oliver B. K. Petersen <nfd708>
Kristoffer A. Kortbæk <jnq578>

Extending Futhark's multicore C backend to utilize SIMD using ISPC

Advisor: Troels Henriksen

Handed in: June 23, 2022

Abstract

This thesis details the design, implementation, and evaluation of a new compiler backend for the data-parallel programming language Futhark, which builds upon an existing multicore C backend. Using the Intel Implicit Single-Program-Multiple-Data Program Compiler (ISPC) language as a code generation target for performance-critical sections of code, we aim to enable the backend to exploit data-parallelism via SIMD instructions, in addition to the existing use of task parallelism, which the backend was designed for. To do so, we extend the existing C code generator to emit ISPC code, which looks very similar to C, but is implicitly vectorized. We describe the details and difficulties of using ISPC as a code generation target, and discuss how suitable of a choice it was.

We evaluate the performance of our backend against the existing multicore C backend, using the extensive suite of benchmarks that ship with the Futhark compiler, and demonstrate that our backend can achieve significant speedups on many of these benchmarks. Our implementation doesn't always yield a speedup, and on several benchmarks, it generates much slower code than the existing multicore C backend. We investigate and describe these deficiencies, and discuss how our work may be improved upon in the future.

Keywords: Implicit parallelism, SIMD, SPMD, Parallel languages, Futhark, ISPC, Advanced Vector Extensions (AVX), Streaming SIMD Extensions (SSE).

Contents

1 Introduction

In this thesis, we detail our work implementing a new compiler backend for the Futhark language compiler, which utilizes the ISPC language to generate code using SIMD (Single-Instruction-Multiple-Data) instructions.

Futhark is a data-parallel language primarily targeting GPUs, but which may also run on multicore CPUs [Hen17] using a backend that emits C code calling the POSIX thread API [Tra20]. The language aims to provide a modern, relatively high level, alternative to writing low-level code using APIs such as OpenCL or CUDA directly. The compiler itself is not parallelizing, and the programmer is responsible for exposing parallel sections of code. This is done concretely by calling a variety of built-in ”array combinators”, which are special-cased functions that the compiler knows how to generate parallel code for. To achieve reasonable performance, the Futhark compiler is aggressively optimizing.

The ISPC language, developed by Intel, is a language based on C, with features and execution semantics specifically tailored towards vectorization using SIMD instructions found in the instruction sets of most modern x86 CPUs [PM12]. It utilizes a ”Single-Program-Multiple-Data” (SPMD) programming model, which allows the programmer to write programs with sequential execution semantics, which are then implicitly run in parallel on multiple pieces of data. Hence, idiomatic ISPC programs often look quite similar to idiomatic C programs, but are implicitly parallel.

Futhark’s multicore C backend currently only directly utilizes task-parallelism via multithreading [Tra20]. In short, parallel operations are divided into smaller chunks, each of which are executed fully sequentially on separate threads. The primary goal of this thesis is to implement and describe a new backend, which extends the multicore C backend, and vectorizes each of the aforementioned sequential operations by emitting ISPC code in place of C code. Effectively, this achieves parallelism at multiple layers, adding data-parallelism with SIMD on top of the existing task-parallelism.

We evaluate not only the performance achieved with such a backend, but also how well suited ISPC is as a code generation target, including descriptions of the challenges we had to overcome, in order to use it effectively. The focus is mainly on providing descriptions and implementations of robust mechanisms for generating ISPC code, and performance is only a secondary concern in this project. Importantly, we limit the scope of the project to deal with *only* the backend portion of the Futhark compiler. Optimization passes earlier in the compilation pipeline, which may have a significant effect on the quality of the code our backend generates, will not be altered as part of our work.

1.1 Contributions

The primary contributions of this thesis are:

- Design and implementation of a new compiler backend for the Futhark compiler, which builds upon the existing multicore C code-generator, but emits implicitly vectorized ISPC code.
- An evaluation of the added backend using Futhark’s extensive suite of benchmarks, some of which are ports of established benchmarks from other projects.

- A port of a handwritten ISPC benchmark, AOBench, to the Futhark language.
- Various improvements to Futhark's suite of tests.
- An extension of the open source Haskell library, `language-c-quote`[1], which enables parsing of ISPC-specific language constructs. This work wasn't utilized in the final backend.

1.2 Overview

The thesis is organized into a few major parts:

- **Part 1 - Background** provides background knowledge relevant to understanding the work described in the rest of the thesis. This includes a brief description of the history of computer architectures, as well as different kinds of parallel execution, and ways to achieve them. Furthermore, the 2 main languages relevant to the thesis are described. These are Futhark, which we are extending the compiler for, and ISPC which is the target language for our compiler backend.
- **Part 2 - Methods** describes various algorithms and implementation methods relevant to our backend, at a theoretical level. These methods lay the foundation of what we are trying to achieve with the backend. Furthermore, we describe a few important details of Futhark's multicore C backend, which we are extending, at a high level.
- **Part 3 - Implementation** describes the concrete implementation of our backend in detail, as well an overview of the compilation pipeline. This part includes a detailed description of how we worked around various issues encountered when trying to programmatically generate ISPC code.
- **Part 4 - Evaluation** contains an evaluation of the implemented backend using various programs from Futhark's benchmark suite. In select cases where the observed results were particularly surprising, we investigate and describe reasons for differences in performance. Additionally, we perform a qualitative evaluation of how suitable ISPC is as a code generation target.
- **Part 5 - Conclusion** is the final part, and contains a brief description of the shortcomings of our work, as well as ideas for what may be improved in the future.

[1]`https://hackage.haskell.org/package/language-c-quote`

Part I
Background

2 Background

This chapter describes the theoretical foundations of our work, including a brief overview of how computer architectures have historically enabled parallel computation.

2.1 Computer architecture

In their simplest form, most modern computer systems can be thought of as having three components: The central processing unit (CPU), main memory, and input/output devices (IO). Data buses carry information between these components. This is the so-called Von Neumann architecture.

Main memory is a temporary storage device that stores both the program to execute and the data that the program manipulates while executing. This storage device is often implemented with a collection of dynamic random access memory (DRAM) chips that only store data for as long as the chip is supplied power [BO15, chapter 1.4].

The CPU is the core of the computer. It executes instructions specified by the program residing in main memory. The instruction set utilized by the CPU could, broadly speaking, be a CISC instruction set architecture (ISA) like x86 or a RISC architecture like ARM. In both cases, the CPU reads the current instruction from main memory, at a location specified by the program counter (PC). The instruction is decoded, and an appropriate operation, dictated by the ISA, is performed, after which the PC is updated. This fetch-decode-execute loop continues for as long as the processor has power supplied [BO15, chapter 1.4]. The program counter itself is a register residing in the ”register file” component of the CPU, which is a very small storage area with a fixed number of registers. The arithmetic logic unit (ALU) performs the actual computations and address calculations.

2.1.1 CPU limitations

The interpretation of Moore’s Law has for a long time been that compute power roughly doubles every two years, while the cost-effectiveness stays the same [Oan18]. However, since 2005, the single-core clock-rate has stagnated. Single-core performance no longer scales, and a program utilizing only a single core will not simply run twice as fast, by purchasing a new CPU to run it on two years later for the same price. Furthermore, the term ”memory wall” was coined due to the once seemingly ever-increasing gap between processor speed and memory speed. Although this gap is closing, due to the still increasing memory speed and stagnating processor speed, it is still an important restriction, since the CPU frequently needs to wait for memory to deliver data for processing. Hardware designers have therefore had to invent new ways to improve performance.

Cache and memory: One way to deal with the memory wall has been to use smaller, faster memory storage devices called ”caches”. These are temporary staging areas for data that the processor might need in the near future [BO15, chapter 1.5]. With these, the CPU can check if the needed data is in the cache, and only fetch more data from the much slower main memory if necessary. Accessing the register file typically takes a single clock cycle, while reading from memory takes more than 100. Usage of a cache bridges the gap between the time it takes to access main memory vs the register file. There are often different layers of cache, where ”L1” cache is the fastest, taking a couple more cycles than the register filer to access, but being much larger than the register file. ”L2” cache is typically slower than L1 cache, but even larger [BO15, chapter 1.5], etc. Although not directly related to cache, modern CPU's often come equipped with specialized hardware known as SIMD units. These are essentially wide registers that can load 128, 256 or even 512 bits of data, and can be used to store vectors of values, as opposed to a single scalar. This way, a single instruction can load more data from memory at once, eliminating the need to wait for the result of multiple scalar loads, which further bridges the gap between processor and memory speed.

Instruction-level parallelism: Instead of fetching single instructions one by one, and executing them in sequence, a modern CPU will fetch several instructions and execute them out of order [Ets13]. A dependency graph is calculated between the instructions, and this enables the processor to run some independent instructions in parallel. Therefore, the fetch-load-execute loop described earlier can be far from what a modern processor actually executes.

2.2 Task and data parallelism

Another, now ubiquitous, way system designers have achieved increased performance is through hardware with parallel capabilities. This includes processors with multiple cores and dedicated SIMD hardware, which enable two different paradigms for exploiting parallelism.

Task-parallelism is a paradigm in which a task is divided into multiple subtasks, each of which are executed in parallel. Concretely, this can be achieved with multithreading or multiprocessing, which are 2 different approaches to utilizing the inherently parallel nature of modern multicore CPU's, making use of threads or processes to distribute work [BO15, chapter 1.9]. The workload is divided among multiple processing units, each of which are free to perform different operations on potentially differing data. In essence, this paradigm uses multiple independent programs working together to achieve a larger goal. In general, task-parallelism is well-suited for algorithms which can be cleanly divided into independent subtasks. Other algorithms might only partially, or not at all, be amenable to task-parallelism, depending on the degree of dependence between each subtask, i.e. the amount of synchronization and thus stalling required to run the tasks in parallel.

Data-parallelism provides a different approach to parallel computation. With this approach, parallelism is exploited by executing the same operations simultaneously on several pieces of data. Vectorizing sequential code, using the increasingly wider SIMD units of modern CPUs, is a practical way to apply this paradigm [KSC$^+$12]. An entire vector of values may be loaded at once, and the same operation performed on each element of the vector. Instead of dividing a task into parallel subtasks, as is done with task-parallelism, data-parallelism divides the data into segments, each of which have the same operations applied in parallel.

Worth noting is that both kinds of parallelism may be used simultaneously. A task can be split into subtasks, each executed by a different thread. Then, in order to exploit more parallelism, each subtask can use vectorization with SIMD in order to achieve a combination of task- and data parallelism.

2.3 SIMD

Vectorization using SIMD instructions is a great way to squeeze more performance out of a program. At their core, SIMD instructions perform the same operation, such as an arithmetic calculation, on multiple pieces of data simultaneously, and are a very direct way to exploit data-parallelism. Concretely, this is done by packing multiple scalars into wide, special-purpose vector registers, which specialized vector instructions may then operate on. There are several ways to make use of SIMD instructions, each with their own pros and cons. We shall briefly describe 2 major methodologies.

2.3.1 Vector intrinsics

Perhaps the most direct way of using SIMD instructions in a program is through the use of ”vector intrinsics”. These are built-in functions that allow the programmer to explicitly vectorize code, by directly wrapping the vector instructions of a specific instruction set. One benefit of such an approach is that it is completely explicit - when the programmer inserts vector intrinsics, the program *will* utilize vector instructions. This allows for very precise control in tuning code for optimal performance. The approach does also have some major downsides, though. Vector intrinsics are inherently bound to a single instruction set, meaning that a program will have to be adapted or rewritten for each new target instruction set, unless an abstraction layer over multiple target instruction sets is used. Additionally, this approach can be tedious and error-prone in practice, and requires the programmer to become quite familiar with the specific instruction set, which may introduce a high level of mental overhead.

2.3.2 Autovectorization

Another approach to vectorizing code is through the use of auto-vectorization. A sufficiently smart compiler can attempt to analyze and automatically identify loops which may be safely vectorized, and transform them to use vector instructions in the loop body. Many major compilers[2,3] implement this approach to some extent. The obvious benefit of such an approach is that it is completely implicit, and the programmer doesn't need to consider how the code may be vectorized at all. Although potentially pleasant for the programmer, auto-vectorization is a very complicated task, especially in languages that support mutation and aliasing of memory, such as C and C++, and the auto-vectorizing capabilities of most compilers are far from perfect.

[2]`https://gcc.gnu.org/projects/tree-ssa/vectorization.html`
[3]`https://llvm.org/docs/Vectorizers.html`

Performing auto-vectorization requires intricate analysis of data dependencies and memory aliasing [Cor12], and there will almost always exist safe to vectorize loops, which the compiler cannot detect. Even simple examples, such as calculating the sum of an array of numbers, can prove tricky for a compiler to automatically vectorize, as such a program will contain data dependencies between loop iterations. These are in general an obstruction to vectorization, even if the specific program may be safe to vectorize. In fact, many compilers, such as Intel's ICC, attempt to automatically detect such ”reduction idioms” and treat them as special cases [Cor12].

These difficulties mean that, in practice, auto-vectorization is very brittle. Since ”auto-vectorization is not a programming model” [Pha19, part 1], the performance of a program is at the mercy of the implementation details of the auto-vectorizer. Performance may be adversely affected by seemingly benign changes, since the programmer has little control over *when* exactly vectorization is performed.

2.3.3 Memory access patterns

Memory access patterns are important to consider when writing programs that utilize SIMD, and can have a large impact on performance, sometimes even making naive vectorization directly detrimental. Generally speaking, there are 2 kinds of memory accesses one can make use in the context of SIMD - contiguous and non-contiguous. When accessing contiguous values in memory, efficient vector loads and stores can be used. This type of instruction reads from or writes to multiple neighboring memory locations simultaneously, taking vector registers as input. In the more general case, when memory locations to access are non-contiguous, ”scatter” and ”gather” operations are used. A scatter operation takes a vector of memory locations and a vector of values, and writes each value to the corresponding location. Similarly, gather operations take a vector of memory locations, and read the values at those locations into a vector register.

Both of these operations are particularly slow on architectures relevant to this thesis, as they necessitate iterating sequentially over each element in the index vector, and because they usually exhibit bad locality of reference. Some newer CPU architectures do, however, come equipped with specialized scatter and gather instructions [Cor11], which may perform better than complete sequentialization of the load/store operations in some cases.

It is worth noting that whether a memory access uses a scatter/gather operation or vector load/store instruction is a statically determined property, as they are 2 fundamentally different types of instructions. There isn't an instruction that will dynamically choose between the 2 types at execution time.

3 The Futhark language

Futhark is a purely functional, data-parallel, statically typed array language, in the ML family [Hen17]. The Futhark language includes many first-order functions, as well as the ability to define more of these, but the data-parallelism is obtained entirely from a selection of ”Second-Order Array Combinators”, or SOACs for short, the four primary of which we describe. These let the programmer write programs in a style similar to other conventional functional languages, while the compiler implicitly generates parallel code to be executed mainly on the GPU via CUDA and OpenCL, or alternatively on a multicore CPU using C. As stated in the introduction, the Futhark compiler does not automatically parallelize programs, and the programmer must use the built-in SOACs to achieve *any* parallelism - the compiler knows specifically how to generate parallel code for them. Thus, the programming model of the language centers around the effective use of these SOACs.

3.1 Second-Order Array Combinators

The following is a description of the four primary SOACs Futhark uses to express parallelism. First, the type of each function is specified. The types are written in a similar style to how they are written in the Futhark language, where $[n]\alpha$ is read as an array of length n and elements of type α. Then, the semantics of the SOACs are described, where $[x_1, x_2, \ldots, x_n]$ represents an array of n elements. Some further requirements may exist for the input, and if so, they will be mentioned as well.

- `map`: $(\alpha \rightarrow \beta) \rightarrow [n]\alpha \rightarrow [n]\beta$
  `map` f $[x_1, x_2, \ldots, x_n]$ = $[f(x_1), f(x_2), \ldots, f(x_n)]$
  The `map` function takes a function f and an array, and returns an array where f has been applied to all elements of the input array.

- `reduce`: $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [n]\alpha \rightarrow \alpha$
  `reduce` f *ne* $[x_1, x_2, \ldots, x_n]$ = $f(\ldots f(f(ne, x_1), x_2) \ldots, x_n)$
  The input array is reduced using the binary operator f. Concretely, this means interspersing a binary operator between each 2 elements on the array, which results in a calculation yielding a single value. For example, `reduce (+) 0 arr` would return the sum of elements in `arr`. The function f must be associative, which means that $f(x, f(y, z)) = f(f(x, y), z)$ for all x, y, z. In addition, ne must be the neutral element for f.

- `scan`: $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [n]\alpha \rightarrow [n]\alpha$
  `scan` f *ne* $[x_1, x_2, \ldots, x_n]$ =
  $[f(ne, x_1), f(f(ne, x1), x2), \ldots, f(\ldots f(f(ne, x_1), x_2) \ldots, x_n)]$
  The result of the `scan` is an array where the element at index i is equal to `reduce` f *ne* $[x_1, x_2, \ldots, x_i]$, for all i where $1 \leq i \leq n$. Therefore, the requirements from `reduce` carry over - f must be associative, and ne is the neutral element for f.

- `reduce_by_index`: $[m]\alpha \rightarrow (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [n]i32 \rightarrow [n]\alpha \rightarrow [m]\alpha$
  `reduce_by_index` *dest f ne is vs*
  The result of this operation can be described in the following pseudocode, where f must be both associative and commutative, meaning that $f(x, y) = f(y, x)$ for all values x, y. In addition, ne is again the neutral element for f.

Listing 1: Imperative pseudocode for `reduce_by_index` [Tra20]

```
for j < length is:
  i = is[j]
  v = vs[j]
  if i >= 0 && i < length dest:
    dest[i] = f(dest[i], v)
```

For each element i in is and a corresponding value v from vs, if i is within bounds, then the i'th index of $dest$ will be updated using f, v, and the previous value at the index. This SOAC can be viewed as a generalized form of `reduce`.

3.2 Fusion

The Futhark compiler is an optimizing compiler, and naturally performs several optimizing transformations to the input source code. Along with typical optimizations such as inlining, copy- and constant propagation, and dead code elimination, one of the steps in this pipeline is fusion [Hen17]. The motivation behind this optimization is to reduce the number of array traversals needed to execute the program, in addition to avoiding the need for time temporary intermediate arrays.

Fusion can be done both horizontally, where two independent SOACs with the same input are combined, and vertically, where several SOACs can be combined into other, often more generalized SOACs. Two examples of the latter transformation are the `redomap` and `scanomap` constructs.

- `redomap` $f\ g\ n\ xs$ = `reduce` $f\ n$ (`map` $g\ xs$)
- `scanomap` $f\ g\ n\ xs$ = `scan` $f\ n$ (`map` $g\ xs$)

These functions both perform a mapping operation followed by either a scan or reduction, but may be implemented with a single loop. In practice, the functions return both the final result of the `reduce` or `scan`, and the result of the `map`, to allow for further fusion. The normal `scan` and `reduce` combinators may be viewed as special cases of `scanomap` and `redomap` respectively, with the identity function used for the function g in the `map`. Therefore, in the following sections, our use of the terms `scanomap` and `redomap` may also refer to regular `scan` and `reduce` combinators which were not subject to fusion. This happens for simple reductions and scans such as `reduce (+) 0 xs`, where there is no `map` available to fuse with the `reduce`.

4 The ISPC language

The Intel Implicit Single-Program-Multiple-Data Program Compiler (ISPC) [PM12] provides a way to utilize SIMD on modern CPUs, that doesn't involve writing vector intrinsics by hand, or relying on brittle auto-vectorization. The compiler accepts a variant of the C programming language with a few notable differences and extra language constructs to support implicit vectorization, and is capable of producing efficient vectorized code targeting either the AVX, SSE, or NEON instruction sets. ISPC is fully interoperable with C/C++, and utilizing it is just a matter of linking an object file against some existing C/C++ code, making it very easy to integrate into existing codebases.

In this section, we describe the semantics and strengths of the language, the programming model that supports them, and some performance pitfalls this model can lead to. We focus on the parts of the language relevant to this project. We shall refer to the source language as ”ISPC”, and to the compiler as ”the ISPC compiler” to disambiguate.

4.1 SPMD on SIMD

ISPC uses a programming model dubbed SPMD-on-SIMD, where SPMD abbreviates Single-Program-Multiple-Data [PM12]. This model is inspired by GPU programming languages, such as GLSL, HLSL and OpenCL, which are well known for their application of SPMD to organize programs that run in parallel on several threads.

With SPMD, the programmer writes a *single program* that runs for each thread, but where each thread works on different data. In a GPU shading language, one may for example write a fragment program that calculates the final color of a single pixel, and then run this in parallel for many pixels. Naturally, programs written to utilize this model often look very similar to their sequential counterparts, as the model implicitly runs many sequential programs in parallel.

In the programming model used by ISPC, the analogue of a thread in the typical interpretation of SPMD is a ”program instance” operating on a single SIMD ”lane”, concretely meaning a section of a SIMD-instruction-enabled register. A group of these program instances is referred to as a ”gang”, with the number of program instances within referred to as the ”gang size”. Unlike with most applications of SIMD to GPU programming, there is only 1 gang per CPU core, and they all share the same instruction pointer. Thus, ISPC does not implicitly utilize task-parallelism. Instead, ISPC programs run in fully sequential lockstep on each core, though the language does provide constructs to support explicit task-parallelism.

4.2 Language semantics

ISPC code looks quite similar to C code, but is implicitly executed in parallel. To illustrate this, consider the following simple C program, which calculates the square of each element in an array of integers:

Listing 2: Simple C program

```c
void squares(int arr[], int size) {
  for (int i = 0; i < size; i++) {
    int val = arr[i];
    arr[i] = val * val;
  }
}
```

An analogous, but parallel, program written in ISPC is shown below:

Listing 3: Simple ISPC program

```
export void squares(uniform int arr[], uniform int size) {
  for (int i = programIndex; i < size; i += programCount) {
    int val = arr[i];
    arr[i] = val * val;
  }
}
```

There are a few differences between the 2 programs, which are worth pointing out. A gang in ISPC can work on 2 kinds of data, ”uniform” and ”varying”. The `uniform` qualifier indicates that data is shared between all program instances, and is equivalent to a scalar value in C. The `varying` qualifier indicates that data varies between each program instance, and is represented internally as a vector of values, with one value per program instance. The default qualifier is `varying`, and can be omitted. Hence, the variables `i` and `val` in the above snippet are varying.

The `programCount` variable used in this snippet is a special uniform constant which denotes how many instances are in a gang. For most architectures, this will be 4 or 8. `programIndex` is a varying integer containing a different scalar integer for each program instance, starting as 0 and counting up in increments of 1. The `export` qualifier simply tells ISPC to make the function callable from C code.

In the program on listing 3, the loop index, which is a varying value, is first initialized to `programIndex`. Using a gang size of 4 for example, this will assign the values `<0, 1, 2, 3>` to `i`, meaning that the variable i will be 0 for program instance 0, 1 for program instance 1, and so on. At each iteration, in parallel, each program instance will load a value from the array into `val`, multiply this value by itself, and write it back into the array. The multiplication will implicitly be implemented with a vector instruction, as will the array reads and writes. At the end of each iteration, we add `programCount` to the loop index, since we operate on that many array elements in parallel during each iteration. At the second iteration, `i` will thus have the value `<0, 1, 2, 3> + 4 = <4, 5, 6, 7>`, so `i` will be 4 for program instance 0, 5 for program instance 1, and so on.

The potential speedup from using ISPC for such a task comes from the fact that each loop iteration works on multiple pieces of data simultaneously. Assuming perfect scaling, one would expect to see a speedup equal to the gang size when comparing to an equivalent sequential, non-vectorized program.

This kind of loop, where we operate on a gang-sized ”chunk” of an array at each iteration, is so common, that ISPC has built-in syntax sugar for it. The listing below shows an alternative, more idiomatic way, to implement the program. This version is semantically equivalent to the one shown on listing 3.

Listing 4: Simple ISPC program with foreach

```
export void squares(uniform int arr[], uniform int size) {
  foreach (i = 0 ... size) {
    int val = arr[i];
    arr[i] = val * val;
  }
}
```

Another, more complex, example is shown below. This time, we calculate the sum of an array of integers.

Listing 5: Sum of array in ISPC

```
export uniform int sumArray(uniform int arr[], uniform int size) {
  int accumulator = 0;
  foreach (i = 0 ... size) {
    accumulator += arr[i];
  }
  return reduce_add(accumulator);
}
```

At each iteration, in parallel, each program instance reads an element from the input array, and adds it to their accumulator. Finally, the scalar accumulators for each program instance are added together using the built-in function `reduce_add`, which sums each scalar in varying value into a single uniform value. Comparing this to an equivalent sequential C program, we again see a striking similarity:

Listing 6: Sum of array in C

```
int sumArray(int arr[], int size) {
  int accumulator = 0;
  for (int i = 0; i < size; i++) {
    accumulator += arr[i];
  }
  return accumulator;
}
```

For cases where there isn't an appropriate built-in cross-lane reduction function available, ISPC provides a `foreach_active` construct, which can be useful for iterating sequentially over scalars in a varying value. For example, listing 5 could be rewritten as:

Listing 7: Example of foreach_active in ISPC

```
export uniform int sumArray(uniform int arr[], uniform int size) {
  int accumulator = 0;
  foreach (i = 0 ... size) {
    accumulator += arr[i];
  }
  uniform int result = 0;
  foreach_active (i) {
    result += extract(accumulator, i);
  }
  return result;
}
```

`foreach_active` iterates over the currently active program instances, returning their indices one by one. Note the use of the `extract` function. This is one of many cross-lane operations provided by ISPC [Corb]. `extract` takes as input a varying value v and a uniform integer index i, and returns a uniform value corresponding to the value of v on the i'th lane. It is used to extract the single scalar value belonging to a specific program instance, from a varying value.

4.3 Execution model

Program execution in ISPC is ”maximally convergent”, which means that program instances in a gang that follow the same execution path will execute each program statement concurrently, and instances that diverge in their path will re-converge at the earliest possible point [PM12]. This simplifies synchronization between program instances drastically, and saves the programmer from having to consider many kinds of common pitfalls in parallel programming. It does, however, have some interesting implications for how control flow is implemented.

During execution, ISPC maintains an ”execution mask”, which is a bit mask with one bit for each program instance. Each bit represents whether the corresponding program instance is currently active and executing, or if it is disabled. When a program instance is disabled, operations on data in its lane of a SIMD register are ”masked out” [PM12]. Control flow that may cause divergence (such as if-statements and for-loops) is implemented by enabling and disabling parts of the execution mask, as illustrated on figure 1.

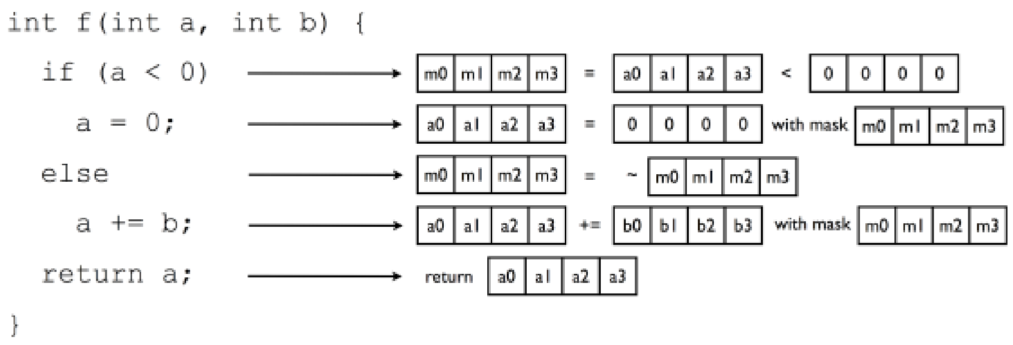

int f(int a, int b) {
if (a < 0)
a = 0;
else
a += b;
return a;
}
m0 m1 m2 m3 = a0 a1 a2 a3 < 0 0 0 0
a0 a1 a2 a3 = 0 0 0 0 with mask m0 m1 m2 m3
m0 m1 m2 m3 = ~ m0 m1 m2 m3
a0 a1 a2 a3 += b0 b1 b2 b3 with mask m0 m1 m2 m3
return a0 a1 a2 a3

Figure 1: Execution of an if-then-else construct showing the execution mask. Figure from [PM12]

The value for `a` is varying, so we might have divergent control flow, where only some program instances enter the first branch; at least conceptually. Since there is only a single instruction pointer, each program instance *will* execute both branches. At the start of the `if`-statement, the mask is set based on which program instances evaluate the condition to `true`. This mask will then be used in the first branch to mask out the assignment to `a` for inactive program instances. When the second branch is reached, the execution mask is inverted. Now, every program instance, which was inactive in the first branch, will be active in the second, and the add-assignment will be masked out for those program instances, which executed the first branch. This method of having every program instance perform every operation, and then masking out unneeded results, effectively treats control flow as data.

Due to this behavior, uniform conditionals will in general be more efficient, since it is ensured that the entire gang will follow the same path, and only one branch has to be executed. In these cases, the compiler will not have to do any mask management and can generate much more efficient code [MdB19]. This is more apparent with computationally heavy branches in a varying control flow. Having a computationally heavy branch causes the inactive program instances to ”stall”. Their part of the execution mask is off, and only values on active vector lanes are used in computation.

4.4 Differences between C and ISPC

ISPC supports most of C’s features, including structs, pointers, enums, macros, and most control flow. However, some features have been adapted to fit the programming model.

Pointers are one feature where the languages differ. Just as with the `const` qualifier in C, the `uniform` and `varying` qualifiers can be applied to both the pointer and the type being pointed to. Thus, all the combinations shown on the snippet below are valid:

Listing 8: All qualifier combinations for a `int *`

```
// a uniform pointer to uniform value, like a C pointer
uniform int * uniform a;
// a vector of pointers to uniform values
uniform int * varying b;
// a uniform pointer to a vector of values
varying int * uniform c;
// a vector of pointers to vectors of values
varying int * varying d;
```

As described in the ISPC user guide [Corb], the default pointer type is a varying pointer to uniform values. However, the result of taking the reference to a varying value is a uniform pointer to varying values. Hence, the following valid C program is invalid ISPC:

Listing 9: Example of difference between C and ISPC

```
int a = 3;
int* b = &a; // compile time error!
```

Since we are attempting to assign a uniform pointer to a varying one. When working with pointers, it is often necessary to add extra typecasts and/or carefully consider which *kind* of pointer is being used.

Another feature where ISPC and C differ somewhat is with structs. In order to support structs that can differ in values per program instance, ISPC implicitly uses a struct-of-array layout for struct valued variables, where each struct contains arrays of values for each field, with the array sizes corresponding to the gang size. This can cause issues when calling C functions from ISPC, which take struct pointers as input. Thankfully, it is fairly straight forward to convert from struct-of-array to array-of-struct [Wal18], as illustrated below:

Listing 10: Conversion from struct-of-array to array-of-struct

```
struct my_struct { int foo; float baz; };

// expose external function written in C
extern "C" unmasked void
  cFunction(uniform my_struct * uniform ptr);

void ispcFunction() {
  my_struct struct_val;                // struct of array
  uniform my_struct aos[programCount]; // array of struct
  aos[programIndex] = struct_val;      // convert
  foreach_active (i) {
    // call C function for each program instance, with each struct
    cFunction(&aos[i]);
  }
}
```

The final major departure from C, which we shall discuss, is function overloading, for which ISPC has full support. This allows the programmer to write multiple versions of a function, each with different input and/or output types, and have the compiler determine which ”overload” of the function to use at each call. For example, consider the following code:

Listing 11: Function overloading in ISPC

```
int add(int lhs, int rhs) { ... }
float add(float lhs, float rhs) { ... }
void test() {
  int a = add(3, 2);
  int b = add(1.3, 3.7);
}
```

On the first line of the body of `test`, the first overload of `add` will be called, and on the second, the second overload will be called, so the `add` function works for both primitive types. Overloading also takes into account type qualifiers, thus the following is valid:

Listing 12: Function overloading with qualifiers

```
uniform float add(uniform float f, uniform float f) { ... }
varying float add(varying float f, varying float f) { ... }
export void test() {
  uniform int a = 1;
  int b = 2;
  uniform int c = add(a, a);
  int d = add(b, b);
}
```

As will be described in coming sections, we have used this mechanism to work around a few challenges faced when generating ISPC code.

4.5 Performance characteristics

In this section, we discuss some important details to keep in mind when trying to write high performance ISPC code.

Due to the implications of all program instances running in lockstep, performance can be greatly affected by declaring variables as `uniform` whenever possible. The ISPC compiler tries to some extent to infer when a varying value could be uniform without altering program behavior, as an optimization, but this breaks down quickly for complex programs [Cora]. Optimizing as much as possible for uniform control flow and non-divergence can be very beneficial.

Something else that can greatly affect performance is the presence of gather and scatter operations as opposed to vector loads and stores. In ISPC, neither gather nor scatter operations are ever written explicitly. Instead, where they appear depends on the manner in which a program is accessing a memory location, and which type is used to represent that memory location. As a consequence, it is easy to unknowingly introduce unnecessary scatter and gather operations when writing ISPC code, so the compiler helpfully emits at warning every time this is the case. We demonstrate a few examples where this might happen:

Listing 13: Examples of different memory loads and stores in ISPC

```
void foo(uniform int * uniform arr, uniform int size) {
  foreach (i = 0 ... size) {
    int a = arr[i];      // vector load
    arr[i] = a;          // vector store
    int b = arr[i * 2]; // gather
    arr[i * 2] = b;      // scatter
  }
}
```

In this program, we see that the first 2 memory accesses can be efficient vector loads and stores, but multiplying the contiguous vector of indices by a constant forces a gather or scatter. Another example is when accessing an array with a varying index which the compiler has no information about:

Listing 14: Example of a load without context

```
void foo(uniform int * uniform arr, int idx) {
  int c = arr[idx]; // scatter
}
```

Even if we only ever pass a value for `idx` which could permit a vector load, this is not something the compiler can guarantee in the general case, so it must always emit a gather. The final example we show illustrates the importance of choosing the right pointer type for memory accesses:

Listing 15: Vector-of-array vs array-of-vector layout

```
void foo(uniform int * varying arr1,
         varying int * uniform arr2,
         uniform int size) {
  for (uniform int i = 0; i < size; i++) {
    int a = arr1[i]; // gather
    int b = arr2[i]; // vector load
  }
}
```

We compare 2 ways of representing an array of integers per program instance. The first choice is a vector of pointers to uniform data. Memory accesses using this kind of pointer, even with a uniform index, necessitate a gather, since each pointer could point to drastically different memory locations. The second choice is a single uniform pointer to varying data. This type of pointer can be accessed much more efficiently with a uniform index, as only a single memory access is required - an entire vector is fetched from a single memory location.

The final consideration we discuss is that of missing vector instructions. The programming model used by ISPC works best when each operation performed maps directly to a vector instruction. Consider the integer and floating point divisions shown below:

Listing 16: Division with different types

```
float a = 5; float b = 3;
float c = a / b;    // fast
int d = 5; int e = 3;
int f = d / e;      // slow
```

On x86 architecture, the floating point division maps nicely to a vector instruction, `vdivps` (vectorized division of packed singles) [Cor16] and can be done very efficiently. For the integer division, on the other hand, there is no such analogous vectorized division. This forces the ISPC compiler to emit code that sequentially performs regular integer division for one program instance at a time [4], slowing down the operation by roughly a factor of the gang size. For this reason, it can sometimes be beneficial to cast data to more appropriate types for vectorization, and then cast the result back, although this may in some cases lead to subtle errors or loss of precision.

[4] `https://ispc.godbolt.org/z/9Wh9jbveW`

Part II
Methods

5 Combining task and data-parallelism

As mentioned in section 3, all parallelism in the Futhark language is expressed in terms of 4 parallel SOACs. In the following sections, we shall describe the existing task-parallel algorithms used by Futhark's multicore backend to implement these combinators. Afterward, we will present vectorized algorithms for these same combinators.

At a high level, each of the 4 SOACs are first and foremost compiled to task-parallel algorithms utilizing multithreading, where each thread is working on its own chunk of the input array. Each thread performs a simple, purely sequential, algorithm on its chunk of the array, and produces some result, which is combined with results from other threads to obtain the final result of the SOAC.

Since each thread performs a sequential algorithm on its own chunk, we can attempt to exploit additional data-parallelism by vectorizing parts of this sequential work. Instead of having each thread performing a purely sequential algorithm, we use ISPC to vectorize. This allows for better total utilization of a multicore processor, and allows us to simultaneously achieve task- and data-parallelism.

6 Task-parallel Algorithms for SOACs

In this section, we describe the currently implemented task-parallel algorithms used by Futhark's multicore backend to compile the aforementioned 4 SOACs. To keep examples simple and readable, we shall use integers in place of what could be any primitive datatype - the SOACs are polymorphic.

6.1 map

The task parallel algorithm for a `map` is very straight forward. A map is simply the process of applying a function f to each element of an array, as indicated by the type signature:

$$\texttt{map} : (\alpha \rightarrow \beta) \rightarrow [n]\alpha \rightarrow [n]\beta$$

Since indices of the array are completely independent, we can just apply the function f to different indices in parallel, and write the result to corresponding indices in the result array, without any special ordering required. In practice, the algorithm splits an input array into roughly equally sized chunks, which are distributed among several threads, each of which performs a sequential map on the given chunk.

6.2 reduce

Recall that reductions intersperse an associative binary operator between each element of an input array, *reducing* the array to a single element. The type signature for reduction is as follows:

$$\texttt{reduce} : (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to \alpha$$

The task-parallel algorithm used for reduction is a fairly straightforward 2-stage process.

- **Stage 1:** In the first stage, the input array is divided into fixed-size chunks. The chunks are distributed to several threads, with each thread working on a single chunk at a time. For each chunk, the reduction of that chunk is calculated sequentially. This reduces each chunk to a single element.
- **Stage 2:** In this second stage, we gather the results of reduction on each chunk from the previous stage into a single, smaller array. A sequential reduction is then performed on this array, on only a single thread. This produces a final reduced element, which is the result of the reduction.

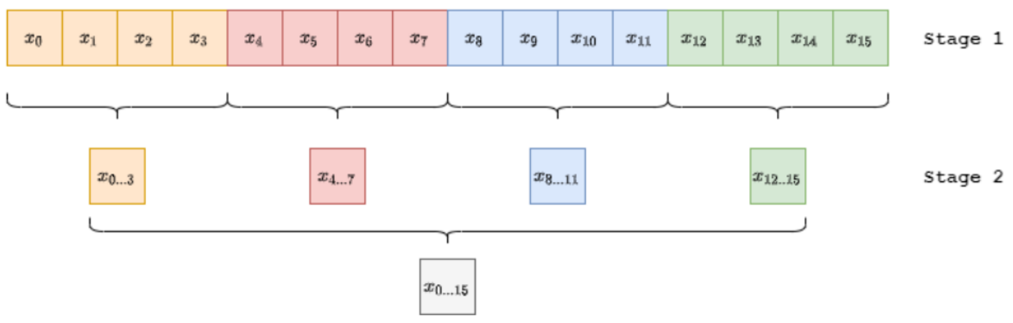


Figure 2: A visual example of the 2-stage reduction algorithm, showing how an input array is split into 4 chunks, each of which are sequentially reduced on separate threads, followed by a single sequential reduction over the results. Figure from [Tra20].

6.3 scan

The task parallel algorithm for `scan` is slightly more complicated, and consists of three separate stages. The type signature for `scan` is as follows:

$$\texttt{scan} : (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to [n]\alpha$$

and the three stages are described below.

- **Stage 1:** In the first stage, the input array is divided into fixed-size chunks, and distributed among threads, just as with `reduce`. Again, a sequential scan algorithm is performed on each chunk. The accumulator used is simply initialized to the neutral element.

- **Stage 2:** In this stage, we perform a sequential scan over the last elements from each chunk of the result array from the previous stage, writing the results back to that same result array. The first chunk uses the neutral element when applying the binary operator. The idea is that the last element of each chunk is the result of a reduction on that chunk, which must be propagated to each subsequent chunk.

  To explain why this is necessary, consider the toy example below, where `scan` is applied to chunks of an array, using a chunk size of 2 elements, with the results written to `res_arr`.

  Listing 17: Scan on toy example

  ```
  -- Desired result of scan is:
  scan (+) 0 [1,2,3,4,5,6] = [1,3,6,10,15,21]

  -- Results of applying scan on chunks and concatenating:
  scan (+) 0 [1,2] = [1,3]
  scan (+) 0 [3,4] = [3,7]
  scan (+) 0 [5,6] = [5,11]
  res_arr = [1,3,3,7,5,11]
  ```

  As shown on the snippet above, *just* concatenating the results from each chunk yields the wrong result. Next, we perform a sequential scan on the last element of each chunk, and write back the results to `res_arr`:

  Listing 18: Scan on toy example

  ```
  -- Do a scan on the last element of each chunk.
  -- Write result back to res_arr at appropriate index:
  scan (+) 0 [3,7,11] = [3,10,21]
  res_arr = [1,3,3,10,5,21]
  ```

  The last element of each chunk in `res_arr` now contains the desired result. However, the remaining elements still need propagation of values from previous chunks, which leads us to the next stage.

- **Stage 3:** This stage is also executed in parallel, with each thread performing a sequential scan on its chunk of the input array, like in stage 1. However, each scan uses the last element of the previous chunk in the *result array* as the starting element, with the first chunk using the neutral element.

  In our previous example, this means using 0, 3 and 10 as the starting elements for each scan. This process is shown below, and calculates the final result of the task-parallel `scan`.

  Listing 19: Scan on toy example

  ```
  -- Result of applying scan to each chunk using a starting
      element
  -- taken from the previous chunk.
  scan (+) 0 [1,2] = [1,3]
  scan (+) 3 [3,4] = [6,10]
  scan (+) 10 [5,6] = [15,21]
  res_arr = [1,3,6,10,15,21]
  ```

6.4 reduce_by_index

The `reduce_by_index` combinator is also known as a generalized reduction, and has the type

$$\texttt{reduce\_by\_index:}[m]\alpha \to (\alpha \to \alpha \to \alpha) \to \alpha \to [n]i32 \to [n]\alpha \to [m]\alpha$$

And is called like so:

```
reduce_by_index dest f ne is vs
```

The function f is repeatedly used to update the value at $dest[is[j]]$ by applying f to the current element in $dest[is[j]]$ and the corresponding element $vs[j]$, where $j \in [0, n-1]$, and out-of-bound indices are ignored. The updated $dest$ is then returned. Worth noting is that duplicate elements may occur in is. The runtime system presented by [Tra20] chooses during execution between two different algorithms - subhistogramming or atomic updates.

6.4.1 Subhistogramming

For this algorithm, a local array of the same size, m, as the original input array is allocated for each thread. Additionally, each thread will get a chunk of the indices, is, and values, vs. Then, each thread works in parallel on its own local array, using its own chunk of is and vs, and any need for synchronization between threads is thus avoided. This produces $n_{subarray}$ local arrays, which have to be combined into one result array. Recall that each local array is m elements long, meaning that we essentially have to reduce $n_{subarray}$ of length m into a single array of length m. This is done with a "segmented reduction", which groups the input elements into segments, that are then reduced over. With this construct, we parallelize over segments rather than individual array elements [Tra20].

Although the sequential reduction on each local array is $\mathcal{O}(n)$, the segmented reduction used to combine the result is $\mathcal{O}(n_{subarray} \cdot m)$ time. Thus, the entire work performed is $\mathcal{O}(n) + \mathcal{O}(n_{subhist} \cdot m) = \mathcal{O}(n_{subhist} \cdot m + n)$, so the algorithm is only work efficient whenever $n_{subarray} \cdot m \leq n$. This is used as a heuristic to only choose subhistogramming when $n_{subarray} \cdot m \leq n$ as mentioned in [Tra20].

6.4.2 Atomic Updates

In this algorithm, all threads are allowed to perform updates on the same global array. To ensure correctness, data races must be eliminated by enforcing atomicity of each operation. The exact manner in which this is done depends on which operator the caller supplies to the `reduce_by_index` call. If the operator can be mapped directly to atomic intrinsics, this is done. Otherwise, the algorithm falls back to a locking-based approach using mutual exclusion [Tra20]. Naturally, this algorithm performs worse the more writes happen to the same index, as the atomicity of writes causes sequentialization.

7 Vectorized Algorithms for SOACs

In this section, we discuss vectorized, SIMD-utilizing algorithms for each of Futhark's main SOACs. These aim to exploit an extra layer of parallelism within each working thread through vectorization. There are, however, cases where we are completely unable to vectorize a SOAC, for example, due to the properties of the supplied inputs. The loss of potential performance compared to sequential execution, should at most be a factor of the SIMD width, which is an acceptable loss. In cases where the SOAC doesn't make sense to vectorize, we may always fall back to generating regular C code which only uses task-parallelism, and doing so adds no overhead.

One notable difference between the algorithms shown here and in the previous section, is that we here present versions of the 4 SOACs that are fused with a `map`. This is also the representation of the SOACs that our back-end is supplied. What was previously a `reduce` or `scan` is now a `redomap` or `scanomap`. These constructs are briefly described in section 3.2.

In the section, we will again use simple integers for examples, and will furthermore use ISPC code as a form of pseudocode for snippets.

7.1 map

Since each iteration of a `map` is independent, it is extremely straight forward to vectorize. Instead of applying the mapping function, which we shall call `map_op`, to each index sequentially, we perform a vector load `arr[i]`, applying `map_op` to the loaded vector, and write back the results to an array, `res`, using a vector store.

Listing 20: ISPC code for data-parallel map

```
void map (uniform int arr[], uniform int res[], uniform int n) {
  foreach(i = 0 ... n){
    int v = arr[i];
    res[i] = map_op(v);
  }
}
```

7.2 redomap

Reductions are a bit more complicated to implement than maps, and can be done so in several ways, depending on properties of the operands. Therefore, in our backend, we use 4 distinct code generation paths which are chosen between at compile time by inspecting the operands. 1 of these 4 paths corresponds to simple sequential, non-vectorized C code, which we will omit describing.

7.2.1 Commutative reduction

The first and perhaps most interesting code generation path is the one for reductions using a commutative binary operator. The commutative property allows for generating particularly efficient code, since the applications of the binary operator can be *interleaved*. This is done concretely by first computing a reduction over segments of the input array in vectorized fashion, which yields a vector as its result, and then sequentially computing a reduction over the elements of that vector, yielding a final scalar. In this way, the algorithm is very similar to the task-parallel 2-stage algorithm for reduction described in the previous section. The vectorized and sequential reduction map nicely to ISPC's `foreach` and `foreach_active` constructs respectively, described in section 4.2.

Listing 21: ISPC code implementing a redomap over an array of integers with a commutative operator.

```
export uniform int reduce_comm(uniform int arr[],
                               uniform int size,
                               uniform int neutral) {
  int vector_accum = neutral;
  foreach (i = 0 ... size) {
    int elem_i = arr[i];
    int mapped = map_op(elem_i);
    vector_accum = reduce_op(vector_accum, mapped);
  }
  uniform int scalar_accum = neutral;
  foreach_active (i) {
    uniform int elem_i = extract(vector_accum, i);
    scalar_accum = reduce_op(scalar_accum, elem_i);
  }
  return scalar_accum;
}
```

Listing 21 shows an implementation of this algorithm. In the `foreach`, we first load a vector of values from the array and apply the supplied mapping function `map_op`. Next, we apply the binary reduction operator, `reduce_op`, on the resulting vector, and the vector accumulator, which is initialized to the neutral element. Subsequently, in the `foreach_active`, we sequentially reduce over the vector accumulator from the previous step, using the `extract` primitive to extract scalar values from said accumulator one by one. The result is stored in a scalar accumulator. Note that `reduce_op` is assumed to work for both scalar and vector inputs, which can be achieved concretely by making several overloads of the function.

It may not be immediately obvious why this operation is interleaved, and why it only works if `reduce_op` is commutative, so to illustrate, we shall describe an execution of the algorithm step-by-step. For this example, we will use a gang size of 4, and perform reduction using addition as the binary operator over the array `[1,2,3,4,5,6,7,8]`, and with the neutral element 0. The mapping function will be the identity function for this example. In other words, our example will be to calculate the sum of the integers 1 to 8.

1. We initialize the vector accumulator to the neutral element, yielding a vector of 0's.

2. We enter the first iteration of the `foreach`, and load the first 4 values of the array into a vector, yielding `<1,2,3,4>`.

3. We add this vector component-wise to our accumulator, yielding `<1,2,3,4>`

4. We enter the next iteration of the `foreach`, and load the next 4 values of the array, yielding `<5,6,7,8>`.

5. We add this vector component-wise to our accumulator, yielding `<6,8,10,12>`

6. We initialize a new scalar accumulator to the neutral element 0.

7. We enter the second stage of the algorithm, where we perform sequential reduction over the values in the vector accumulator, loading them 1 by 1 and adding them to a scalar accumulator. The scalar accumulator's value after the first iteration will be 6, after the second 6+8=14, after the third 14+10=24, and finally 24+12=36. Thus, 36 is the final result.

In step 3 of this example execution, we broke the order of operations of the reduction. Semantically, reduction with an associative operator is equivalent to inserting a binary between each 2 elements of an array. For this example, that would yield the expression `1+2+3+4+5+6+7+8`. However, what our algorithm has actually computed is the value of the expression `(1+5)+(2+6)+(3+7)+(4+8)`. This interleaved reordering of terms is only possible because of the commutative property of addition. This is what allows for utilizing vector as opposed to scalar addition, which maps directly to efficient SIMD instructions, making this code generation path by far the most desirable for performance.

7.2.2 Non-commutative reduction

In cases where the binary operator used for reduction is not commutative, we have no choice but to fall back to performing the reduction purely sequentially. We can, however, still vectorize part of the SOAC - namely the mapping function. Luckily, non-commutative reductions are fairly uncommon, and when they do occur, the associated mapping function is usually non-trivial, making it worth vectorizing.

Listing 22: ISPC code implementing a redomap over an array of integers with a non-commutative operator.

```
export uniform int reduce_noncomm(uniform int arr[],
                                  uniform int size,
                                  uniform int neutral) {
  uniform int scalar_accum = neutral;
  foreach (i = 0 ... size) {
    int elem_i = arr[i];
    int mapped = map_op(elem_i);
    foreach_active (j) {
      uniform int elem_j = extract(mapped, j);
      scalar_accum = reduce_op(scalar_accum, elem_j);
    }
  }
  return scalar_accum;
}
```

Listing 22 illustrates this code generation path. Again, the algorithm starts by initializing an accumulator to the neutral element, but this time the accumulator is a scalar. In the `foreach`, we load a vector of values from the array, and apply our mapping function `map_op` to that vector. Then, in the nested `foreach_active`, we sequentially apply our reduction operator `reduce_op` to the accumulator and values of the vector returned from the mapping function. With this algorithm, *only* the initial loading of values from the array, and the application of the mapping function are vectorized, but if the mapping function is sufficiently expensive, this can yield decent speedup. Furthermore, the loads of values from the array map to efficient vector-load instructions, which is beneficial regardless of the supplied mapping function.

7.2.3 Reduction with mapped operator

The final code generation path is for a special case of reduction, where a ”mapped operator” is supplied. A mapped operator consists of a stack of nested `maps`, with a binary operator at the innermost level. An example of a program using a reduction of this nature is shown on listing 23.

Listing 23: Program using a mapped operator for reduction.

```
def main [n][m] (as: [n][m]i32): []i32 =
  reduce (map2 (+)) (replicate m 0) as
```

Such a pattern will generate a stack of nested for-loops, with one additional for-loop per level of nesting of the input array. In the snippet above, we are reducing over an array of arrays of scalars, which creates 2 for-loops, as illustrated by listing 24. Note that the multidimensional array is flattened to one dimension in the generated code.

Listing 24: ISPC code implementing a redomap with a mapped operator.

```
export void reduce_mapped(uniform int arr[], uniform int res[],
                          uniform int n, uniform int m,
                          uniform int neutral) {
  // Initialize an array-valued accumulator to neutral element
  for (uniform int k = 0; k < m; k++) {
    res[m] = neutral;
  }
  // Perform the reduction, with the innermost loop vectorized
  for (uniform int i = 0; i < n; i++) {
    foreach (j = 0 ... m) {
      int elem_ij = arr[i * m + j];
      res[j] = reduce_op(res[j], elem_ij);
    }
  }
}
```

To utilize SIMD for this construction, we simply turn the innermost `for`-loop into a `foreach`-loop, again as illustrated by listing 24. This is safe to do, since the mapped operator, which is just a stack of nested `maps`, will have no dependencies between loop iterations. Essentially, we are vectorizing the *operator itself*, rather than the reduction loop. Note that there is no mapping function shown in the snippet above, since this pattern prevents the compiler from fusing the reduction with an adjacent `map`.

7.3 scanomap

`scanomap`, like `redomap`, also has different code generation paths depending on the properties of the operands supplied. However, due to the near-unavoidably sequential nature of a `scan`[5], the vectorized algorithm only has three distinct code generation paths. Again, like for `redomap`, one of these code generation paths corresponds to sequential and non-vectorized code generation, which will not be described further.

[5]We present a more efficiently vectorized algorithm for scan in section 12.2

7.3.1 Vectorizing the mapping function

Like non-commutative reductions, scans exhibit data dependencies between loop iterations, which forces us to perform it sequentially. However, we can still vectorize the mapping function of a `scanomap`.

Listing 25: ISPC code implementing a `scanomap` over an array of integers.

```
export uniform int scan_stage1(uniform int arr[],
                               uniform int res[],
                               uniform int size,
                               uniform int neutral) {
  uniform int scalar_accum = neutral;
  foreach (i = 0 ... size) {
    int elem_i = arr[i];
    int mapped = map_op(elem_i);
    foreach_active (j) {
      uniform int elem_j = extract(mapped, j);
      uniform int scan_res = scan_op(scalar_accum, elem_j);
      res[i] = scan_res;
      scalar_accum = scan_res;
    }
  }
}
```

In listing 25 we see the vectorized algorithm for a `scanomap`. A scalar accumulator is initialized with the neutral element. The `foreach` loop then loads a vector `elem_i`, to which we apply the mapping function. Then, the nested `foreach_active` sequentially applies the binary operator to elements in the vector of mapped elements, `mapped`, and the accumulator. Thus, the `scan` portion of the `scanomap` is still computed sequentially, with only the `map` portion vectorized.

Recall that the task-parallel algorithm for `scan` consists of 3 separate stages, each of which perform a scan operation, with the second stage being performed only on one thread. We note that, since there isn't much performance to be gained from vectorizing this middle stage[6], we only vectorize the first and third stage using the algorithm provided above.

7.3.2 Scan with mapped operator

As for with reductions, scans may be passed a mapped operator, which we handle in precisely the same manner as is described in section 7.2.3. The performance of this approach is further evaluated for the case of `scan` in section 10.7.

7.4 reduce_by_index with fused map

The `reduce_by_index` SOAC is not very straightforward to vectorize, and we argue that the benefit from doing so would be fairly negligible. In this subsection, we will discuss the problems we encountered with the SOAC, which, as mentioned in section 6.4, has 2 distinct code paths.

[6]The middle stage will only scan over an amount of elements equal to the number of cores. This is effectively just a constant.

7.4.1 Subhistogramming

Recall that with the subhistogramming approach, the input arrays of values and indices are chunked and distributed among multiple threads, with each thread working on its own local sub-array. This algorithm could perhaps have been extended to utilize the SIMD model, by using a local array not only for each thread, but also for each SIMD lane. However, we have not prioritized implementing this, due in part to the following concerns:

1. This SOAC is rarely used compared to the others.
2. The memory accesses will in general be incoherent, leading to scatters and gathers, which tend to bottleneck performance.
3. As mentioned in [Tra20], when the size of each local array is large, the overhead of initializing them can take longer than just performing a sequential algorithm. Utilizing SIMD will increase this initialization overhead by up to a factor of the gang size. The heuristic used to choose between the subhistogramming and atomic updates approach would likely have to be updated to account for this overhead.
4. To store a local array per SIMD lane, we would have to allocate a factor of the gang size more memory, in comparison to the regular sequential version of the algorithm.

For these reasons, we don't vectorize the majority of the code generated for `reduce_by_index` with this approach. However, if the `reduce_by_index` is fused with a `map`, we do vectorize this, in the same way as for `redomap` and `scanomap`. Additionally, we vectorize the segmented reduction used by `reduce_by_index`, simply by turning the outermost loop, which iterates over segments, into a `foreach`.

7.4.2 Atomic Updates

The atomic update approach is also not ideal for performance in the context of SIMD. The 2 first problems mentioned in the previous subsection about subhistogramming, also apply to this approach. Additionally, if one program instance stalls on an atomic operation, then by nature of ISPC's execution model with a shared program counter, all other program instances must wait for the atomic operation to finish. On a more practical note, atomics with 16-bit values are still not supported, meaning that some atomics would have to be performed in C. We could certainly make this work, due to the easy communication between C and ISPC as mentioned in subsection 9.3. However, due to these issues, we decided to not generate ISPC code for the atomic updates approach.

Part III

Implementation

8 Compiler overview

In this section, we will give an overview of the compilation pipeline implemented by the Futhark compiler. The pipeline is divided into 3 sections, which we call the front-end, middle-end and back-end.

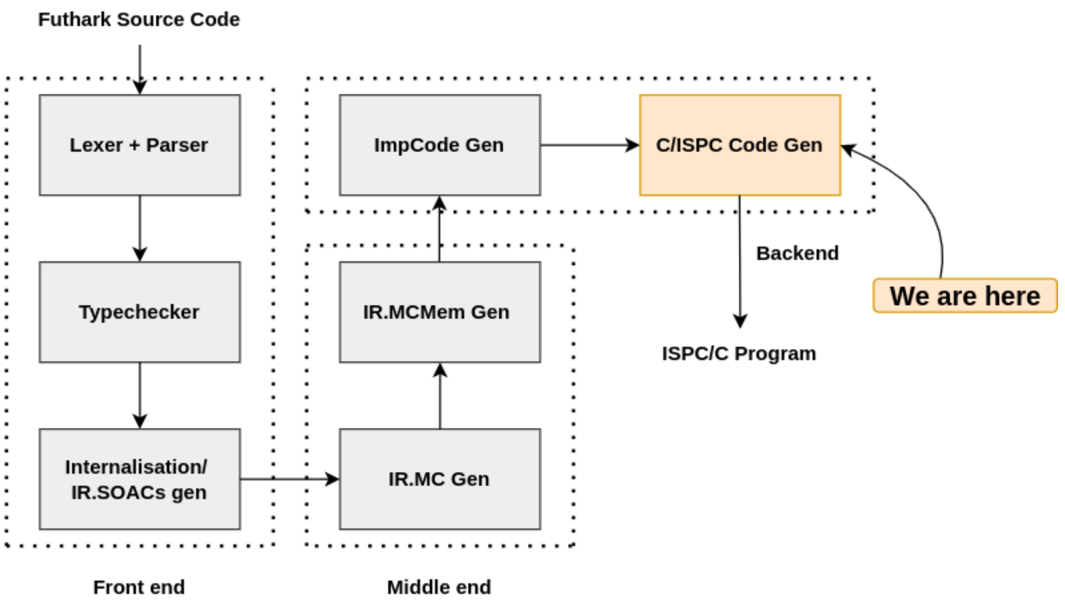


Figure 3: Overview of the full compilation pipeline used for the ISPC backend

8.1 Front-end

The primary responsibility of the front-end is to do lexing, parsing, and type checking of the Futhark source language. Additionally, it also has an internalization stage, where the Futhark source language is transformed to a ”Core IR” (IR abbreviating intermediate representation) parameterized with a ”SOACS” representation. This process includes defunctionalization to first-order functions, among other transformations, with the goal of making later program transformation easier. The Core IR is passed to the middle-end.

8.2 Middle-end

The middle-end works on programs using the Core IR representation. Depending on the target back-end and specific optimization stage in the middle-end, programs can use different instantiations of the parameterized Core IR. In general, the middle-end compilation pipeline for the multicore backend uses the following instantiations:

$$\text{SOACS} \longrightarrow \text{MC} \longrightarrow \text{MCMem}$$

In the following sections, we will briefly describe the pipeline stages associated with each of these intermediate representations.

8.2.1 SOACS Generation

SOACS is the representation provided by the front-end after internalization. Here, the notion of a SOAC is introduced to the Core IR language. Optimizations such as dead binding removal, common subexpression elimination, inlining and fusion (mentioned in section 3.2) use this representation.

8.2.2 MC and MCMem Generation

For the multicore and ISPC backends, the SOACS representation is transformed into ”MC”, which can represent multicore computation on the CPU. Two things are worth noting about the representation of various SOACs at this stage. Firstly, a SOAC in this IR is a stack of perfectly nested `maps`, with an innermost computation[7]. This could be scalar computation, `redomap`, `scanomap` or `reduce_by_index`.

Secondly, when programs reach the MC IR stage, two semantically equivalent versions of each SOAC are generated, each exploiting different levels of parallelism. These are:

1. A version where every nested SOAC has been sequentialized. This version only exploits task-parallelism at the outermost level.

2. A version which tries to exploit nested parallelism, by leaving all nested SOACs intact.

In this context, sequentializing a SOAC means to convert each task-parallel loop used by SOACs nested within into regular, sequential `for`-loops. Both versions will, however, still be converted to code utilizing data-parallelism via SIMD. The multicore backend is equipped with a small runtime, that includes a ”scheduler”, which, among other things, chooses between the 2 versions of each SOAC at runtime. Whenever it is time to execute a SOAC, the scheduler will decide which to use based on factors such as the problem size and overhead of creating threads [Tra20]. This allows the generated program to exploit nested parallelism until there is no longer a benefit in doing so.

Finally, the MCMem IR expands upon MC with explicit memory allocations. This is the last representation used before backend code generation. Since the MC/MCMem IR was originally designed for the multicore backend, it only explicitly represents task-parallelism, and has no notion of vectorization with SIMD. It is out of the scope of this thesis to extend the middle-end with optimizations targeted towards utilization of SIMD, although this may be fruitful.

[7]This is alluded to and further described in section 7.2.3.

8.3 Back-end

The majority of the work in this thesis relates to the back-end of the compiler (see figure 3). The primary responsibility of the backend is to generate code in the target language, which is a mixture of C and ISPC in our case. As a stepping stone for generating concrete source code, the compiler first generates an imperative intermediate language, named ”ImpCode”. Thus, we compile each SOAC represented in the MCMem IR to ImpCode implementing the concrete parallel algorithms described in sections 6 and 7. A good chunk of our contribution was the implementations of these algorithms at the ImpCode level. The remaining bulk of our work was in generating actual ISPC code, which we describe in further detail in the section about code generation, section 9.

8.4 Scheduling strategies

We've briefly mentioned how the scheduler chooses between multiple versions of a SOAC at runtime. As the name implies, the scheduler additionally has the responsibility of scheduling and later executing these SOACs. When doing so, the scheduler uses one of 2 strategies [Tra20]:

- **Static Scheduling:** The workload is shared evenly among threads, and this division of work is never updated for the SOAC being executed.
- **Dynamic Scheduling:** The workload is divided into a pool of many small chunks, which various threads may take from and execute. The size of each chunk varies dynamically during execution, based on the time spent on previous execution of chunks.

The strategy used by the scheduler for each SOAC is decided at compile time, and is based on the irregularity of the workload. For regular, evenly spread workloads, static scheduling is used, but for workloads which may be irregular, dynamic scheduling is used. Such irregularity may occur when the body of a SOAC contains a loop without a static bound (i.e. a `while`-loop).

The implementation of the dynamic scheduling approach was problematic for our backend, as the chunk size used would frequently be 1 loop iteration per chunk. Since the vectorization our backend adds exists within each chunk, such a small chunk size will result in vectorization providing *no* benefit whatsoever, since there aren't multiple iterations to execute in parallel using SIMD instructions. This results in lackluster performance when the approach is used. To alleviate this, we have changed the scheduler to use a minimum chunk size equal to the number of SIMD lanes in the target instruction set, such that we always get the full benefit from vectorization.

9 Code generation

In this section, we describe the implementation of the portion of our backend which generates C and ISPC code. This includes descriptions of how we chose to work around challenges faced when generating ISPC code programmatically.

9.1 Extensions to ImpCode

As stated earlier, a large chunk of our contribution to the Futhark compiler lies in the final stages of code generation, where the imperative ImpCode representation is generated and converted to concrete code in the target language. As such, we have extended the ImpCode representation with a few new constructs, which allow ISPC code to be represented. Concretely, these additions are data constructors representing:

- An ISPC ”kernel”, which is the entry point into ISPC code generation. We use ”kernel” to describe an ISPC function which is exposed to C code, and which implements a *closure*. As such, each kernel is called from exactly one location in C code, and has access to all the variables that are in scope at the call site.
- A `foreach` loop.
- A `foreach_active` loop.
- A combined variable assignment and `extract` call, used for extracting a scalar value from a vector.

To produce ISPC code, one must use the ImpCode representation of a kernel, which may contain more ImpCode that can use ISPC-specific constructs, such as `foreach` loops. Each of the 4 basic kinds of SOACs are compiled to intermediate code implementing one of the algorithms mentioned in section 6 and 7, and using the ImpCode representation. The ImpCode is then passed to the appropriate final code generator for the backend, which emits either C or ISPC code. As mentioned earlier, the exact manner in which the SOACs are translated to ImpCode relies on various properties of the specific SOAC in question, such as the commutativity of the binary operator in the case of reductions.

This extended intermediate representation is used both for the regular multicore backend, and our ISPC backend. Thus, the existing backend needs to handle the newly added ImpCode constructs. This is done by treating regular C code generation as a special case of vectorized code generation with a gang size of 1.

Listing 26: Comparison of ISPC (Left) and C (Right) generated from the same intermediary representation

```
foreach (i = 0 ... n)
{
  // ...
}
foreach_active (i)
{
  // ...
}
a = extract(b, i);
```

```
for(int64_t i = 0; i < n; i++)
{
  // ...
}
int64_t i = 0;
{
  // ...
}
a = b;
```

Listing 26 depicts how some added constructs map to ISPC and C respectively, for the 2 different backends. The pure C equivalent of the ISPC kernel construct, which isn’t depicted, is simply to inline code that would have gone into the kernel, embedding it directly into the call site. Since each kernel is a closure, all the free variables in the inlined code exist in the enclosing scope, so this is safe to do.

9.2 An ISPC code generator

Although ISPC is based on C, there are numerous, often subtle, differences between the 2 languages. Some notable differences are described in the ISPC user guide [Corb] and include:

- No access to the C standard library.
- Different built-in numerical types.
- No string constants or arrays of characters as strings.
- No `goto` statements inside varying control flow, such as a `foreach` loop.
- No union types.
- No volatile qualifier.
- Slightly different semantics for pointers and typecasts.

Additionally, ISPC adds plenty of features to the C language which are not valid C code, some of which are described in section 4. For these reasons, we have opted to produce a new ISPC code generator, instead of fully reusing the code generator shared by Futhark's other C-based backends. This code generator accepts the ImpCode representation and generates a syntax tree which is later pretty-printed. In the interest of avoiding redundant code, this new code generator falls back to the existing C code generator in certain cases where there is no divergence between the C and ISPC code corresponding to a language construct in the intermediate representation. Some code duplication was unavoidable, however.

9.3 Communication between C and ISPC

Perhaps the most important requirement for our backend to function is a robust mechanism for communicating between C and ISPC code. Thankfully, C and ISPC code can be linked together easily (see section 4), and we need only specify exported and external function definitions to do so.

Each ISPC kernel at the intermediate code level maps directly to an ISPC function which is made visible to C code using the `export` qualifier, and is called from exactly 1 place in C code. The prototypes for these functions are directly inserted into the main C source file. As mentioned in section 9.1, these kernel functions implement closures, and have access to all variables bound in the scope they are being called from. This is implemented by each kernel function taking as an input a pointer to an ad-hoc generated struct, with one field for each free variable in the body of the kernel. The names of each free variable stay identical to those used at the call site. This technique simplifies the boundary between C and ISPC, and allows us to generate code for kernel bodies as if they were directly embedded in the calling C code.

Listing 27: Declaration of an ISPC kernel in C

```
struct futhark_mc_param_struct {
  struct futhark_context *ctx;
  int64 free_var;
  unsigned int8 *free_as_mem;
  unsigned int8 *free_mem;
};
int futhark_mc_loop_ispc(int64 start, int64 end,
  struct futhark_mc_param_struct *futhark_mc_param_struct_);
```

In programs with nested parallel operations, ISPC kernels may need to use the runtime system to schedule additional work, which necessitates calling back into C code. In order to make this process simple, and prevent having to expose most of the runtime system's machinery to ISPC, we generate an ad-hoc C function in these scenarios, which interacts with the runtime system. This function is forward declared in ISPC code and called from a kernel. These ad-hoc C functions are a recurring pattern, which we employ whenever we want to perform an action from ISPC code which is either infeasible or impractical, such as when we need to do string manipulation for the purposes of error handling.

Listing 28: Example of an ad-hoc C function for scheduling in ISPC

```
extern "C" unmasked uniform int futhark_mc_schedule_shim(
  uniform struct futhark_context *uniform ctx,
  uniform struct futhark_mc_task *uniform args,
  uniform int iterations);
```

9.4 Variability analysis

One major challenge in generating ISPC code is getting the variability of variables correct throughout a generated program. Failure to emit code with appropriate variability causes several issues. A loss of performance may occur, with an example being normal scalar memory access turning into scatter or gather operations when a variable, that could have been uniform, isn't marked as such. Additionally, incorrect variability may cause compilation errors, with an example being attempting to assign a varying value to a uniform variable, which is illegal in ISPC.

In order to tackle this issue, we present a simple algorithm for inferring the variability of variables throughout the program based on their uses, which works well in practice, and prevents the compiler from having to take variability into account until the last parts of code generation. The implemented algorithm is reminiscent of how one might implement a mark-and-sweep garbage collector, and describing it will be the focus of the remainder of this section.

The algorithm is implemented in the final phase of code generation, and takes the ImpCode intermediate representation as input. For each variable used at the ImpCode level, we perform a *dependency analysis* and use the results of this to infer the variability of the variable. The dependency analysis takes as input a section of ImpCode and produces as output a mapping from variable names to sets of variables names, with one entry in the mapping per variable used in the code. The associated sets of variable names specify the names of dependencies for each variable.

We handle 2 kinds of dependency: Direct data dependencies, and dependencies caused by control flow. Direct data dependencies occur when the value of a variable may directly depend on the value of another variable, such as is the case when assigning one variable to another. Control flow induced dependencies may occur whenever a variable is assigned within a piece of conditional control flow, be it a `while`, `for` or `if` construct. An example to illustrate this kind of dependency is given on listing 29. Even though the variables `foo` and `bar` are never used directly within the same statement, `foo` must depend on `bar`. Specifically, if `bar` is varying, so must `foo` be, since divergence between program instances in the conditional will necessarily result in a varying value for `foo`.

Listing 29: Snippet illustrating a control-flow induced dependency.

```
int foo = 0;
if (bar) {
  foo = 1;
} else {
  foo = 2;
}
// the value foo at this point depends on bar
```

At a high level, the implemented algorithm performs the following steps:

1. The algorithm starts execution when we are about to generate ISPC code from a section of ImpCode, i.e. a kernel.

2. Find the ”roots” of varying variability within the kernel. These are the symbolic indices of `foreach` loops as well as names of memory blocks declared within the kernel[8].

3. Calculate the immediate dependencies (such as variables appearing on the right-hand side of an assignment) of each variable used within the kernel, including dependencies caused by control flow.

4. Iterate the immediate dependencies to a fixed point, yielding for each variable a set of transitive dependencies.

5. For each variable used in the code, use the associated set to check if the variable depends on any roots of varying variability. If so, the variable is marked as `varying`, and otherwise, it is marked as `uniform`.

The calculation of a fixed point is necessary since the input code is not in static-single-assignment form[9]. The treatment of memory blocks as roots of varying variability is due to every memory block declared within ISPC being varying, and representing a memory block per program instance. Any memory reads from these memory blocks will result in varying values.

[8]Taking local memory blocks into account is necessary since reads from them will produce varying values. For brevity, this isn't discussed further.

[9]`https://en.wikipedia.org/wiki/Static_single_assignment_form`

Listing 30: Example for variability analysis.

```
int a = 0, b = 0, c = 0, d = 0;
foreach (i = 0 ... 100) {
  b = a;
  a += i;
  if (b < 3) {
    c = 2;
  }
  d += 1;
}
```

To further illustrate the algorithm, we shall run it on a simple pseudocode program shown in listing 30. The first step is to find the roots of variability, which in this case is just the symbolic index of the `foreach` loop, `i`. Next, we traverse the code and calculate the immediate dependencies of each variable, yielding the results shown in table 1. Notice that `c` depends on b, which is a control flow induced dependency, while the rest are direct data dependencies. Next, we calculate the fixed point of these immediate dependencies, yielding the results shown in table 2.

Table 1: Immediate dependencies

Variable	Dependencies
a	i
b	a
c	b
d	

Table 2: After iteration to fixed point

Variable	Dependencies
a	i
b	a, i
c	b, a, i
d	

Finally, for each variable used, we check if that variable depends on any roots of varying variability. This is the case for variables `a`, `b` and `c`, but not for variable `d`. Thus, every variable except for `d` would be given the `varying` qualifier in the generated ISPC code, and `d` would be given the `uniform` qualifier.

9.4.1 Coherent memory access

As mentioned in the beginning of this section, one of the reasons variability analysis is necessary, is in order to prevent unnecessary scatter and gather operations. [PM12] refers to memory accesses with varying indices, but which can be turned into efficient vector loads or stores, as *coherent*, and the ISPC compiler implements a late optimization pass that attempts to detect memory accesses of this kind in order to produce more optimal code. This optimization is in many cases critical for good performance, as scatter and gather operations are much slower than vector loads and stores. In our experience, this optimization pass isn't perfect, and occasionally misses cases where a programmer with sufficient knowledge would be able to infer that a scatter or gather is not necessary. Additionally, we found that properly annotating the variability of values used in the program, and in particular annotating everything that *can* be uniform as such, improves the ISPC compilers ability to detect varying memory accesses which need not be gathers or scatters. In this way, our variability analysis algorithm can be viewed as a method for ensuring, that the optimization pass, implemented by the ISPC compiler, receives the most optimal input we can provide, given the intermediate representation being translated.

9.5 Memory allocations

In order to translate most non-trivial programs, our backend needs to handle memory management within ISPC code. In Futhark's C-based backends, there are 2 kinds of allocated memory, each with their own associated API.

Reference-counted memory is the default kind, and is represented as a struct containing a raw pointer, a size in bytes of allocated memory, and a reference counter. C functions are provided to facilitate allocating, unreferencing and assigning to and from memory of this kind, each of which update the reference counter accordingly. When the reference counter reaches 0, the memory is deallocated.

Lexical memory is only used within the lexical scope it is declared in, or in any nested scope, and is always deallocated immediately at the end of the scope it is declared in. This type of memory is represented as a pair of a raw pointer, and an integer representing the size in bytes of allocated memory. The classification of lexical memory is purely an optimization to lessen allocator pressure and avoid the slight overhead of reference counting.

The API for interacting with both kinds of memory has been exposed to ISPC via forward declarations, but this isn't sufficient for handling all cases of allocation. Since ISPC code implicitly runs for multiple program instances simultaneously, operations on memory declared in ISPC, such as allocation, copying and deallocation, should happen not just once, but once *for each* program instance. To facilitate this, we have made use of function overloading. Figure 31 shows an example of this, in which we have 2 overloads of the function `memblock_alloc`, which performs allocation of reference counted memory. The first of these is a forward declaration, where the implementation is in C code, while the second is written in ISPC. The first takes as input a uniform pointer to a uniform struct representing the memory block, which implies a single allocation, while the second takes a uniform pointer to a varying struct, which implies an allocation for each active program instance. We use ISPC's `foreach_active` construct to sequentially allocate for each active program instance, and make use of the trick shown in listing 10 to convert between struct-of-array and array-of-struct, which is necessary when calling C functions from ISPC which take struct pointers as input.

Listing 31: 2 overloads of a function used for memory allocation.

```
extern "C" unmasked uniform int memblock_alloc(
    uniform struct futhark_context * uniform ctx,
    uniform struct memblock * uniform block,
    uniform int64 size,
    uniform const int8 * uniform block_desc);

static uniform int memblock_alloc(
    uniform struct futhark_context * uniform ctx,
    varying struct memblock * uniform block,
    uniform int64 size,
    uniform const int8 * uniform block_desc) {
  uniform int err = 0;
  varying struct memblock _block = *block;
  uniform struct memblock aos[programCount];
  aos[programIndex] = _block;
  foreach_active(i) {
    err |= memblock_alloc(ctx, &aos[i], size, block_desc);
  }
  *block = aos[programIndex];
  return err;
}
```

9.5.1 Array-of-vector vs vector-of-array

When declaring and allocating regions of memory in ISPC, a choice must be made about the memory layout. When we want to allocate a block of memory for each program instance, we have 2 choices of pointer type to refer to this memory - either a vector of pointers to uniform data, or a uniform pointer to varying data. We shall refer to these 2 layouts as vector-of-array (VOA) and array-of-vector (AOV) respectively.

Both layout choices have pros and cons. The default (without any explicit annotations) is VOA, as it is what is most compatible with the semantics of sequential C code, and most C code that uses pointers directly can be translated almost verbatim to ISPC when using this layout. Unfortunately, due to each program instance having its own pointers to distinct locations in memory, any memory accesses with this layout become inefficient scatter and gather operations, even when the memory offset used is uniform.

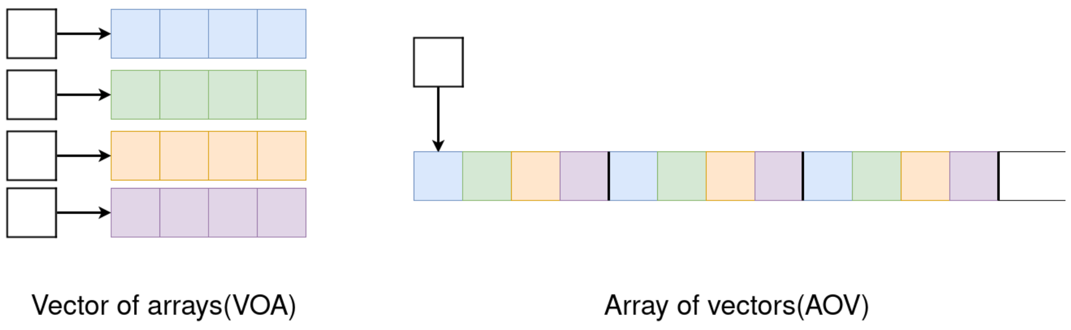


Figure 4: Vector-of-array layout compared to array-of-vector layout for a gang-size of 4. Each program instance has a distinct color to highlight its own chunk of a single array, in case of AOV layout, or its own allocated array in case of VOA. White boxes represent pointers.

When using AOV instead, such memory accesses become efficient vector loads/stores, which is further illustrated in figure 4. Furthermore, locality of reference, and thus cache usage, may in some cases be improved for gathers and scatters when using this layout, since the memory locations accessed are more likely to be close together than with VOA.

This does come with some downsides, though. This layout uses a single contiguous block of memory, which is shared between all program instances, and the values belonging to each program instance are interleaved, as illustrated in figure 4. Thus, using this layout means that allocations become "all or nothing" - there is no such thing as only a single program instance allocating memory. In that way, using this layout is analogous to the transformation performed by Futhark's *memory expansion* pass [Hen17]. In situations where only a few of the program instances need to perform a memory allocation, much of the allocated memory will be wasted. Additionally, certain operations, such as copying memory from one location to another, require special handling with this layout, due to the interleaved nature of values belonging to each program instance. This means that we cannot use ISPC's built in `memmove` or `memcpy` operations, but need to manually copy over elements one by one. The interleaved property also causes issues when calling functions that expect as input a uniform pointer to an array of scalars. With the VOA layout, we can simply pass the pointer belonging to one program instance, but with AOV, there is only pointer shared among all program instances, which is pointing to data with an incorrect layout. This would force us to manually de-interleave the array prior to performing the call, and copy back the data one element at a time, in case the call changed any of the elements in the passed array.

Due to the outlined issues with AOV layout, we limit our attempts at using the layout to those blocks of memory which are *only* used within a single ISPC kernel. Here, we reuse the notion of lexical memory blocks mentioned in section 9.5. Lexical memory declared in an ISPC kernel will by definition only be used in that kernel, so it is safe to change said memory to use AOV layout.

9.6 Error handling

Handling of errors originating from within generated ISPC turned out to be a somewhat unexpected challenge. Quite a few changes had to be made to our first attempt at ISPC code generation, in order to allow for effective error handling. Unlike many languages designed for high-performance computing, Futhark is able to produce friendly error messages at runtime, even when targeting the GPU. This includes catching arithmetic errors such as divisions by 0 and negative integer exponentiation, as well as preventing out-of-bounds memory accesses [Hen20]. Additionally, failure to allocate or free memory causes an appropriate error message.

Listing 32: Simplified example of a C function which may produce an error

```c
int foo (struct futhark_context* ctx) {
  int err = 0;
  // allocate some memory
  struct memblock* my_mem;
  if (memblock_alloc(ctx, &mem, 100, "my_mem") != 0) {
    err = 1;
    goto cleanup;
  }
  // perform some operation that may fail
  if (operationThatMayFail(ctx) != 0) {
    err = 1;
    goto cleanup;
  }
  // ... more code goes here
  // free any allocate memory
  cleanup:
  if (memblock_unref(ctx, &my_mem, "my_mem") != 0) {
    return 1;
  }
  return err;
}
```

Error handling is implemented in a fairly straightforward manner. Most functions that may produce errors have a similar structure to the one shown in listing 32. There are a few key elements to point out here:

- Functions that may fail return an error code, which is just an integer.
- Early outs are used whenever an operation has failed. Concretely, `goto` statements are used to jump to a cleanup section which frees allocated resources and then returns.
- A global context object is passed around to each operation. This object contains, among other fields, a string containing a relevant error message. String literals are passed around, which are used to construct these error messages.

Immediately, there are some issues with translating this directly to ISPC code, the main one being harsh restrictions on `goto` statements in ISPC, and another being the complete lack of strings and string literals. We tackle these issues one by one.

9.6.1 Lack of strings and string manipulation

ISPC has no string type, no string literals, and no built-in functions for performing operations on strings. Many of the functions in Futhark's runtime system expect to be supplied with string literals to produce better error messages. We solve this problem by replacing each occurrence of a string literal in generated ISPC code with a call to an ad-hoc generated C function, which simply returns the appropriate literal as a `char*`. This lets us completely circumvent the lack of string literals, at the cost of overhead from calling into C functions that cannot be inlined. This overhead is, however, only present in code paths which aren't particularly performance-critical, such as after a fatal error has occurred. Generally speaking, the performance delta is quite minimal.

A similar approach is used to deal with the lack of utilities for string manipulation. For example, some error handling code involves concatenating several strings together to form a complete, customized error message. In these cases, we simply move the entire body of code performing the problematic string concatenation into another ad-hoc generated C function, which takes as inputs the necessary values to perform the concatenation. Using these approaches, we manage to keep ISPC code that produces error messages quite similar to the pure C equivalent, while calling into tiny ad-hoc C functions whenever necessary. Errors such as checks for out-of-bounds memory accesses and divisions by 0 are handled using this approach. When generating code in a fairly restricted language, such as ISPC, it is convenient to have the ability to make ”escape hatches” like this, and the language thankfully makes this very simple to do.

9.6.2 No general mechanism for early-outs

ISPC has `goto` statements, but only allows them to appear in uniform control flow, making them essentially unusable for the style of error handling we have implemented. Other mechanisms for exiting a function early, such as `break` and of course `return` statements, exist, but also have restrictions imposed on them. Notably, both `break` and `return` are not permitted within any `foreach` or `foreach_active` loops. These restrictions unfortunately almost completely prevent early-outs in idiomatic ISPC code.

To work around these issues, we have implemented a separate code generation path for ISPC kernels which can handle errors. This code generation path does not use `foreach` or `foreach_active` statements, which are both syntax sugar for much more verbose code, as shown in listing 33.

Listing 33: Expressing a vectorized loop in ISPC with and without `foreach`. Not using the syntax sugar leads to more code due to duplication of the body.

```
// With "foreach" syntax sugar:
foreach (idx = 0 ... n) {
  // some code
}

// Without "foreach" syntax sugar:
for (uniform int i = 0; i < n / programCount; i++) {
  int idx = programIndex + i * programCount;
  // some code
}
if (programIndex < (n % programCount)) {
  int idx = ((n / programCount) * programCount) + programIndex;
  // some code, same code as the previous scope
}
```

Even with these changes, `goto` statements are still not feasible to use, as they are not permitted in varying control flow, which occurs quite often. `return` statements, on the other hand, are completely legal anywhere outside of vectorized loops. However, using `return` directly in place of `goto` would cause resource leakage - as illustrated in listing 32, `goto` isn’t just used as an early return, but also to free resources before returning.

This issue lead to splitting each ISPC kernel into 2 distinct parts; an outer ”wrapper” kernel used for resource management, and an inner ”worker” function, which contains business logic. Each outer wrapper has exactly the same interface described in section 9.3, being a simple closure which takes as input a pointer to a struct containing free variables.

The wrapper is responsible both for some memory allocations, and for freeing all resources allocated within both the worker function and wrapper itself.

The inner worker function takes as input not only a pointer to the same struct of free variables, but also a pointer to a separate struct containing pointers to all the memory allocated in the outer scope. Essentially, the worker function is also a closure, nested within another closure. This approach allows the worker function to simply `return` whenever an error occurs, letting the outer wrapper take care of freeing resources. The resulting behavior is very similar to what the `goto` statements from the previous example were doing.

Listing 34: Simplified example of code generation path supporting early-outs for error handling.

```
// Ad-hoc struct with free variables
struct free_struct {
  uniform int free_foo;
};
// Ad-hoc struct with declared memory
struct mem_struct {
  varying struct memblock * uniform mem_bar;
};

// Inner wrapper function only called within ISPC
static inline uniform int inner_worker(
    uniform int start, uniform int end,
    uniform struct free_struct * uniform fs,
    uniform struct mem_struct * uniform ms) {
  // Put free variables into local variables
  uniform int foo = fs->free_foo;
  // Put declared memory into local variables
  struct memblock mem_bar = *ms->mem_bar;
  { /* ... business logic, may return early */ }
  // Write back potentially changed memory
  *ms->mem_bar = mem_bar;
  // If we reach the end, return success
  return 0;
}

// Outer wrapper kernel exposed to and called from C
export uniform int outer_wrapper(
    uniform int start, uniform int end,
    uniform struct free_struct * uniform fs) {
  // Put free variables into local variables
  uniform int foo = fs->free_foo;
  // Allocate some memory
  struct memblock mem_bar = alloc(foo, ...);
  // Build the struct containing allocated memory
  uniform struct mem_struct ms;
  *ms.mem_bar = mem_bar;
  // Call the worker function
  uniform int err = inner_worker(start, end, fs, &ms);
  // Read back potentially changed memory
  mem_bar = *ms.mem_bar;
  // Free allocted memory
  free(mem_bar);
  return err;
}
```

Listing 34 shows a simplified example of some code generated with this approach. There are a few practical things to take note of. First, we unpack the struct containing free variables twice, once in the outer wrapper and once in the inner. The unpacking in the outer wrapper is due to the fact that allocations may make use of these free variables, for example to calculate the allocation size. Another noteworthy piece is the write-back of memory passed to the inner worker function. In cases where memory from the outer wrapper is changed via reallocation or reassignment in the inner worker function, these changes must be communicated back to the outer wrapper, such that the correct resources are freed in the end. This write-back should occur before any `return` statement is executed.

This alternative code generation path is capable of handling errors effectively, but that comes at a cost. The size of the generated code is much larger, and overhead is introduced by the repeated packing and unpacking of structs containing free variables. Therefore, when we are about to generate code for an ISPC kernel, we traverse the body of the kernel in order to determine if the kernel may produce an error or not. In cases where the kernel cannot fail, we don't take this code path, instead producing a single ISPC function which may use `foreach` loops. In cases where errors may occur, we prioritize correctness and enable the use of early-outs, and in cases where no errors can occur, we prioritize performance, emitting more idiomatic code with less overhead.

9.7 Automatic binding generation

Futhark's C-based backends occasionally generate ad-hoc built-in functions for the runtime system, with types and number of arguments varying for each function. Some situations where this may occur are when annotating user functions with the `#[noinline]` attribute, or when using certain built-in functions, such as the one for matrix transposition. Some of these built-in functions are polymorphic in Futhark, and may be called in several places with different types of inputs. Therefore, in the multicore backend, C-code for these functions is generated ad-hoc when they are used. An ISPC kernel may call these functions, and it is therefore necessary to support them in ISPC. This problem can be tackled in two ways:

1. The functions could be generated directly in ISPC. We could potentially utilize SIMD and generate more efficient code than the C equivalent.
2. We could reuse the generated C code for each function and expose this to ISPC by creating bindings.

We opted for the latter of the 2 options, and will briefly describe each of them to justify that decision.

In the first approach, we would have to generate two different versions of each function to handle both varying and uniform control flow. The advantage is that these functions could be optimized for ISPC, although the implementations are more tricky to produce.

The second approach would require us to call into C. This surfaces a new problem - ISPC does not allow one to pass structs by value to an external function. Only struct pointers may be passed, so we would have to generate wrappers in C that accept pointers. Again, it is necessary to generate ISPC functions that accept both varying and uniform arguments.

Listing 35: Implementing transpose wrapper in c

```c
int builtinmap_transpose_i32_extern(
  struct memblock *destmem_ptr, ...) {
  return builtinmap_transpose_i32(*destmem_ptr, ...);
}
```

In the listing 36, we show ISPC-inspired pseudocode exposing a particular built-in matrix transposition function to ISPC. `builtinmap_transpose_i32` is called from ISPC, and accepts a struct by value, but immediately takes a pointer to it, and passes it to `builtinmap_transpose_i32_extern`, which is a C function wrapping the actual built-in we are trying to expose. This wrapper is shown on listing 35.

Listing 36: Implementing transpose wrappers in ISPC

```
extern "C" unmasked uniform int builtinmap_transpose_i32_extern(
  struct memblock *destmem_ptr, ...);

uniform int builtinmap_transpose_i32(
    varying struct memblock destmem, ...) {
  // This little dance is SOA <-> AOS conversion
  uniform struct memblock vos_destmem[programCount];
  vos_destmem[programIndex] = destmem;
  uniform int err = 0;
  foreach_active(i) {
    err |= builtinmap_transpose_i32_extern(&vos_destmem[i]);
  }
  destmem = vos_destmem[programIndex];
  return err;
}

uniform int builtinmap_transpose_i32(
    uniform struct memblock destmem, ...) {
  return builtinmap_transpose_i32_extern(&destmem, ...);
}
```

The advantage of this second approach is that it is easier to implement than the alternative, just requiring us to generate a few additional thin wrappers around existing C code. We suspect that the potential performance gains from vectorization would be rather negligible, since use of these bindings is fairly uncommon.

9.8 Math primitives

Futhark's runtime has an extensive collection of mathematical operations that are implemented for each backend. The implementations are usually just wrappers around corresponding implementations from each target language's respective standard library. However, ISPC's standard library is still not mature in some ways.

A larger amount of manual work was needed to implement the various built-in math functions, than for with other backends. This is due in part to the need for a separate version of each library function for both uniform and varying input. Furthermore, some relevant ISPC standard library functions were not implemented for all sizes of floats or integers, in which case error-prone casting was necessary.

When the ISPC standard library does not have a function required for implementing a math primitive, it can instead be imported from C and called sequentially, as seen in listing 37. In a few special cases, the function can instead be implemented as a specialized vectorized algorithm.

Listing 37: Using a C library function in ISPC via a wrapper.

```
extern "C" uniform double tgamma(uniform double x);
double futrts_gamma64(double x) {
  double res;
  foreach_active (i) {
    uniform double r = tgamma(extract(x, i));
    res = insert(res, i, r);
  }
  return res;
}
```

9.9 Atomic operations

Although the atomic updates approach to `reduce_by_index` is never generated in ISPC, other compiled programs still make use of atomic operations in parallel loops. ISPC provides built-in atomics, but these are slightly different from those found in GCC. This includes the naming of the atomic operations and the semantics of the atomic compare-and-swap. We were able to solve this discrepancy between the atomics through preprocessor macros, and we could therefore reuse the multicore implementation of atomic operations. As an example, we will show how we used macros to implement an ISPC function that is semantically equivalent to GCC's atomic compare-and-swap.

Listing 38: Using macros to have the same interface for atomics as GCC provides.

```
// Mimic GCC's signature for atomic compare-and-swap in ISPC.
#define __atomic_compare_exchange_n(x,y,z,h,j,k)
    atomic_compare_exchange_wrapper(x,y,z)

// Use macros to create wrappers for multiple types.
#define make_atomic_compare_exchange_wrapper(ty)             \
static inline varying bool atomic_compare_exchange_wrapper(  \
    uniform ty * varying mem,                                \
    varying ty * uniform old,                                \
    const varying ty val){                                   \
  varying ty actual = atomic_compare_exchange_global(        \
                          mem, *old, val);                   \
  bool res = 0;                                              \
  if(actual == *old){                                        \
    res = 1;                                                 \
  } else {                                                   \
    *old = actual;                                           \
  }                                                          \
  return res;                                                \
}
make_atomic_compare_exchange(int)
make_atomic_compare_exchange_wrapper(float)
...
```

For starters, the atomic is in ISPC is called `atomic_compare_exchange`[10], and has a slightly different type signature than in GCC, so we first have to simulate this in the macro definition. Where the atomic operations will differ next is in their respective return values.

[10]The equivalent version in GCC would be called `__atomic_compare_exchange_n`

ISPC will always return what was stored in `ptr` before the atomic was performed, and GCC will return a boolean, with `true` indicating successful comparison. Again, in order to reuse the multicore back-end code generation of atomics, we simulate the GCC behavior in ISPC by comparing the returned result of the atomic operation with the value of `*old`. In case they match, the operation must have succeeded. This way we get the same interface for the atomic operation in both GCC and ISPC.

9.10 Use of language-c-quote

The Futhark compiler relies upon an open-source library called language-c-quote, which utilizes Template Haskell[11] to embed lexing and parsing of various C dialects directly into Haskell code. The library works by invoking a C parser, written with Yacc[12], at compile-time, and embedding Haskell code to produce a C syntax tree, before finishing compilation.

Listing 39: Inlined C code in Haskell using language-c-quote

```
createSumFunction :: String -> C.Exp -> C.Definition
createSumFunction func_name bound =
  [C.cedecl|
    int64_t $id:func_name(int64 arr[]){
      int64 x = 0;
      for(int64 i = 0; i < $exp:(bound); i++){
        x += arr[i];
      }
      return x;
    }|]
```

Listing 39, shows an example of utilization of language-c-quote in Haskell code. We make use of the `[C.cedecl|...|]` construct, also known as an expression quotation. Everything between the `|`'s must be syntactically valid C code. The library builds ”templates” from these invocations, which may have values injected into them, and will produce a C syntax tree when evaluated. In the above example, the `bound` expression and the function name may be injected, as indicated with the `$` symbol, which is called ”splice”. The prepended `C.cedecl` denotes that the resulting C code is a declaration. The library supports various other C syntax elements, such as expressions, types etc.

Listing 40: Inlined C code in Haskell using language-c-quote

```
// Given the Haskell code:
createSumFunction "sumFirst10Elements" [C.exp|10|]

// Produce a syntax tree representing (roughly) this program:
int64 sumFirst10Elements(int64 arr[]) {
  int64 x = 0;
  for (int64 i = 0; i < 10; i++){
    x += arr[i];
  }
  return x;
}
```

[11] `https://wiki.haskell.org/Template_Haskell`
[12] `https://en.wikipedia.org/wiki/Yacc`

Listing 40 shows how calling the `createSumFunction` function with parameters `"sumFirst10Elements"` and the C expression `10`, produces a syntax tree for a syntactically valid C function called `sumFirst10Elements`, which sums the first 10 elements of an input array.

Early in our work with this thesis, we developed an extension to language-c-quote, which adds support for ISPC[13]. During this work, we realized that ISPC was further from being a true dialect of C, than we had initially expected. This made it tricky to extend the library with ISPC support, without either interfering with other C dialects, or introducing hacks, which the maintainers presumably would not have allowed into the library. Therefore, we have instead chosen to make use of language-c-quote's ”escaped statements”, which allow the user to insert raw text into a template, bypassing lexing and parsing entirely. Using this approach allowed us to use ISPC-specific features with no additional changes to Futhark or language-c-quote. While this counteracts some benefits of language-c-quote, and makes the templates slightly less readable, it seemed like the most viable option given the scope of the thesis.

[13] `https://github.com/pema99/language-c-quote/tree/ispc`

Part IV
Evaluation

10 Benchmarks

In this section, we shall describe the results of running a representative subset of the benchmarks in Futhark's benchmark suite. We primarily compare execution times between our ISPC backend and the regular multicore C backend, upon which it is based. In some cases where the performance of our backend is notably worse than expected, we investigate and attempt to provide explanations. Since we present a rather large subset of the benchmarks, we will not investigate every benchmark result in detail, instead focusing on some of the most interesting ones.

CPU	Intel i7-8700 @ 3.2GHz
GPU	NVIDIA RTX 2070 Super
RAM	16 gb, 2400 MHz
OS	Windows 10
Target ISA	avx2-i32x8

Table 3: Specification of the setup used to carry out all benchmarks.

An important detail to note, is that we have rounded benchmark time measurements to the nearest whole millisecond for readability, but speedups are calculated with the measurements *before* rounding. Therefore, the speedups may appear to not match the stated measurements.

10.1 Benchmarking tools

In this subsection, we briefly describe the tools used for carrying out the benchmarks.

10.1.1 Futhark bench

The Futhark language ships with a small tool for reproducible benchmarking[14]. We utilize this tool, `futhark-bench`, in all of our benchmarking. The tool handles every step of the benchmarking process, from compilation of the source program, to running the compiled binary with the relevant input.

10.1.2 Profiling with `gprof`

In certain cases, it is insightful to profile binaries generated by the Futhark compiler in order to gain an understanding of why a given benchmark performs significantly worse (or better) than expected. For this, we use the GNU tool `gprof`[15]. The GCC compiler is passed the

[14]`https://futhark.readthedocs.io/en/latest/man/futhark-bench.html`
[15]`https://linux.die.net/man/1/gprof`

`-pg` flag to generate profiling information, while both the GCC and ISPC compilers are fed the `-g` flag to generate debugging information. When the compiled program is run, a file with profiling information is generated. This is fed to `gprof` in order to produce an overview of how much time was spent executing each line of source code. With this method, we can see timings with line numbers for both ISPC and C code.

10.1.3 Debugging

In cases where a generated program behaves unexpectedly, which is the case for a few Futhark's benchmarks, we typically use either the GNU debugger, `gdb`, or the general analysis tool `valgrind`[16].

10.2 Accelerate

The Accelerate benchmark suite contains Futhark ports of various benchmarks originally written using Accelerate[17], which is a Haskell DSL. These are mostly fairly small toy programs, usually producing some visual output. First, we present the subset of this suite used in [Tra20].

Benchmark	Input		Multicore (ms)	Ours (ms)	Speedup
crystal	2000		422	58	x7.22
	4000		1668	229	x7.30
mandelbrot	2000x2000		45	20	x2.27
	4000x4000		175	72	x2.43
	8000x8000		703	276	x2.54
nbody	10000		62	7	x8.45
	100000		5746	607	x9.47
smoothlife	128		181	133	x1.36
	256		544	248	x2.19
	512		2198	832	x2.64
	1024		10525	3633	x2.90
tunnel	2000x2000		535	58	x9.27
	4000x4000		2118	223	x9.49
	8000x8000		9183	903	x10.17

Table 4: Benchmark results from the subset of the Accelerate benchmark suite used in [Tra20].

As is evident from table 4, the results observed in this subset of benchmarks are generally quite good. The programs generated by our backend are faster than those generated by the regular multicore C backend in all the shown cases. In some cases, we reach or even exceed the maximum expected speedup of 8x from vectorization using 8 SIMD lanes. In

[16] `https://valgrind.org/`
[17] `https://github.com/AccelerateHS/accelerate-examples/`

general, we observe that the speedup achieved from using our backend increases as a function of the program input size. Calling between C and ISPC code with our setup naturally introduces some constant overhead (and prevents the compiler from inlining code), which is more noticeable for smaller input sizes.

Scheduler strategy		**Ours (ms)**
Minimum chunk size = 1		2120
Minimum chunk size = gang size		276
Static scheduling		450

Table 5: Performance of mandelbrot benchmark with various scheduler strategies

The mandelbrot program serves as a nice demonstration of the effect of our changes to the scheduler (see section 8.4). This program utilizes dynamic scheduling, which was initially problematic for our backend, but was later beneficial after we changed the minimum chunk size, as illustrated by table 5. Simply allowing dynamic scheduling to be used, without making any changes to accommodate vectorization, would have been a huge loss of performance.

Benchmark	**Input**		**Multicore (ms)**	**Ours (ms)**	**Speedup**
nbody (Barnes-Hut)	10000		21	25	x0.83
	100000		84	149	x0.56
fluid	medium		23	31	x0.73
pagerank	random_medium		80	106	x0.76

Table 6: Benchmark results from the subset of benchmarks from the Accelerate suite, which had worse performance.

We did also, for some benchmarks, observe a slowdown with our backend, as shown in table 6. Important to note is that each of the benchmarks shown run for a particularly short amount of time. As mentioned above, the overhead of using the generated ISPC code plays a role, and could account for some performance discrepancy. That being said, the nbody (Barnes-Hut) program does seem to scale negatively with input size, which suggests a more fundamental performance issue.

Analysis showed that most of the time for nbody (Barnes-Hut) is spent in a particularly nasty `while`-loop, which, due to unfortunate placement of an enclosing `foreach` (see section 12.1 for details), performs a gather operation in its body. Additionally, the loop often runs for only a few active program instances, stalling the others. The time spent in this loop seems to scale with the input size, hence the negatively scaling performance. The bad performance of the fluid benchmark is similarly explained by unfortunate `foreach` placement, causing unnecessary gathers and scatters nested deeply within several loops.

Worth noting is that the pagerank benchmark is slowed down significantly by integer modulo operations, for which there is no vector instruction, and for which our implementation is especially slow due to handling of edge cases, as described in section 11.2.1. We noticed that over half the execution time is spent performing modulo operations.

[Tra20] found slowdowns when comparing the multicore C backend's performance to the reference Haskell implementations of the crystal and tunnel benchmarks. Interestingly, with our ISPC backend, these slowdowns have been eliminated, and the programs now exhibit substantial speedups over the reference implementations.

10.3 Micro

The Micro benchmark suite is a set of benchmarks specifically designed to test the performance of various Futhark language constructs and intrinsic functions, such as the 4 SOACs described in section 3. This doesn't include benchmarks for the `map` SOAC, since it is very simple, and exercised by almost all other benchmarks. We only include the largest problem size for each benchmark (usually $n = 10^8$), since the smaller problem sizes execute too fast to meaningfully measure performance.

Since we only vectorize `map`, commutative `redomap` and the mapping function of `redomap`, `scanomap`, or fused `reduce_by_index`, the performance characteristics we expect to observe may seem a bit unintuitive. The benchmarks in this suite are designed to test individual SOACs in isolation, so the mapping functions used by these SOACs are very simple, and we only expect to see a significant speedup for commutative reductions. In other words, an ideal outcome for most of the presented benchmarks is to match the performance of the regular multicore C backend. We investigate the benefit of vectorizing mapping functions in section 10.7.

10.3.1 reduce

Table 7 shows the results for the various benchmarks of the `reduce` SOAC. As expected, commutative reductions (the first 4 rows) exhibit speedup. Interestingly, the speedup is much larger for sum_iota_f32 and sum_iota_i32 than for sum_f32 and sum_i32. All these benchmarks calculate the sum of a large range of numbers, but the former variants do not read these number from an array, instead using the loop index directly. By profiling the benchmark, we found that the performance discrepancy is explained by the execution time being dominated by reading values from memory in the latter benchmarks - addition operations are generally very fast. In this case, the speedup from using vector instructions for addition is much greater than the speedup for using vector loads instead of reading scalar values from memory one-by-one.

Benchmark		**Multicore (ms)**	**Ours (ms)**	**Speedup**
sum_iota_f32		10	2	x6.58
sum_iota_i32		4	2	x2.86
sum_f32		17	15	x1.17
sum_i32		14	13	x1.07
prod_mat4_f32		6	6	x0.99

Table 7: Benchmark results for a subset of the reduce micro benchmark

The prod_mat4_f32 benchmark was included as it is an example of non-commutative reduction, computing matrix multiplications. As we had hoped, the performance of our backend matches the performance of the unmodified multicore backend in this case. Interestingly, we found that the exact structure of the `foreach` loops used in the commutative reduction benchmarks mattered a lot for performance.

Listing 41: 2 ways of structuring foreach loops

```
// Old, slow version:
foreach (pre_idx = 0 ... n_18341) {
  int idx = start + pre_idx;
  ...
}
// New, fast version:
foreach (idx = start ... start + size) {
  ...
}
```

By changing the structure of emitted code slightly, as illustrated on listing 41, we observed a x10.8 speedup on the sum_iota_f32 benchmark. We assume this is simply due to the ISPC compiler not catching a possible optimization, although it is surprising.

10.3.2 scan

The results of the micro benchmark results for `scan` are shown in table 8. As we had hoped for, the performance of the SOAC seems to match that of the multicore back-end.

Benchmark		Multicore (ms)	Ours (ms)	Speedup
sum_iota_f32		69	69	x1.0
sum_iota_i32		67	69	x0.96
sum_f32		100	105	x0.95
sum_i32		103	99	x1.03
prod_mat4_f32		4	4	x1.08

Table 8: Benchmark results for a subset of the scan micro benchmark

The performance of prod_mat4_f32 is interesting, since we see a small increase in performance. This could be explained by the small, but non-trivial, mapping function of the `scanomap`, or by the compiler using vector loads when fetching data from memory. Although the rest of the `scanomap` has been fully sequentialized, this is apparently enough to yield some speedup. Of course, the speedup could also just be due to noisy time measurements, since the execution time of the program is very fast, but we did manage to reproduce it on several executions.

10.3.3 reduce_by_index

Below are the results for the micro benchmark testing `reduce_by_index`. As for with `scan`, there isn't much to comment on here. The performance of the multicore and ISPC backend is very similar, as we had expected and hoped for.

Benchmark		Multicore (ms)	Ours (ms)	Speedup
sum_i32		3	3	x1.01
sum_f32		4	4	x1.05
sum_vec_i32		1	1	x1.01
abs_i32		4	4	x0.98

Table 9: Benchmark results for a subset of the reduce_by_index micro benchmark

10.4 Rodinia

The Rodinia Benchmark Suite is a set of benchmarks designed for benchmarking heterogeneous computing infrastructure, written in OpenCL, OpenMP and Cuda [CBM$^+$09]. Futhark ships with ports of these benchmarks, which we investigate in this section.

Benchmark	Input		Multicore (ms)	Ours (ms)	Speedup
LUD	512		10	16	x0.67
	2048		242	213	x1.14
BFS (heuristic)	64k		35	43	x0.81
	1M		64	72	x0.89
K-means	8/204800		251	263	x.95
	5/494019		331	411	x0.81
lavaMD	3-boxes		6.44	8.22	x0.78
	10-boxes		298	101	x2.9
backprop	4000		203	123	x1.65

Table 10: Benchmark results from the subset of the Rodinia benchmark suite used in [Tra20], plus a few extras.

From the benchmarks shown in table 10, we see a mixed set of results. LUD and lavaMD both give speedup on their largest problem sizes, and slowdown for the smaller problem sizes. Backprop, on the other hand, exhibits decent speedup, while BFS and K-means always exhibit slowdown.

Analyzing the performance of the K-means benchmark reveals that most of the execution time is spent on gathers. Roughly 80% of the CPU time is spent on memory reads during two different invocations of `reduce_by_index`, both using subhistogramming. Again due to bad placement of a `foreach`-loop with nested `for`-loops, as mentioned in section 12.1, the memory addresses are calculated by multiplying a number onto the varying loop index, which forces a gather.

The lavaMD benchmark consists of mostly `maps`, and should therefore be easy to vectorize. It does contain nested parallelism, however, and suffers from the same issue as the K-means benchmark. For the largest problem size, the non-optimal memory accesses are outweighed by the performance gained from vectorizing `maps`. For the small problem size, 3-boxes, each kernel invocation only executes 1-3 loop iterations of the outermost `foreach`. This makes vectorization almost irrelevant, since we never utilize the full 8 available SIMD lanes. With this problem size, the non-optimal memory accesses are no longer outweighed, and result in a slowdown.

Just as with the previously described benchmark, LUD is negatively affected by bad `foreach` placement. There is, however, also another section that bottlenecks performance. The generated code for LUD contains a stack of nested loops with uniform loop indices, all within a `foreach`. These nested loops are executed completely sequentially. Removing boilerplate code, the section looks like so:

Listing 42: Lud

```
foreach(SegMap_i = 0 ... n) {
  for(uniform int i1 = 0; i1 < m1; i1++) {
    for(uniform int i2 = 0; i2 < m2; i2++) {
      for(uniform int i3 = 0; i3 < m3; i3++) {
        for(uniform int i4 = 0; i4 < m4; i4++) {
          float x1 = mem.mem[i4 + i3 + i1];
          float x2 = mem.mem[i2 + i4];
          float ret = x1 + x2;
          // ...
        }
      }
    }
  }
}
```

This code leads to repeated sequential memory access, for which there is no obvious gain from using ISPC. However, use of AOV layout, described in section , had a large effect on the performance of this code. With VOA layout, the memory accesses shown in listing 42 become gathers, while with AOV, they become vector loads. After implementing AOV layout, we noticed a decent speedup for both problem sizes of the benchmark.

Benchmark	**Input**		**VOA (ms)**	**AOV (ms)**	**Speedup**
LUD	512		22	18	x1.23
	2048		337	220	x1.51

Table 11: Benchmark results comparing different array representations with the LUD benchmark

This switch to AOV layout made our implementation faster than the regular multicore backend for the largest problem size. As seen in the table above, the performance delta between the 2 layouts is quite significant.

10.5 FinPar

The FinPar benchmark suite is a set of benchmarks designed for testing performance of parallel financial calculations [ABB$^+$16]. They are originally written in C/C++ by the HIPERFIT group, but have since been ported to Futhark.

Benchmark	Input		Multicore (ms)	Ours (ms)	Speedup
LocVolCalib	small		250	355	0.71x
	medium		422	138	3.05x
	large		11312	3365	3.36x
OptionPricing	small		87	215	0.40x
	medium		110	182	0.61x
	large		1644	1134	1.45x

Table 12: Benchmark results from the FinPar benchmark suite

Table 12 yet again shows speedup scaling with the problem size. For the largest input of the LocVolCalib benchmark we see a significant performance improvement, but for small input sizes, too few iterations are performed to negate the overhead introduced by our backend, barely providing any speedup.

For the OptionPricing benchmark, we don't observe any speedup when using anything but the largest input size. Interestingly, the execution time for the medium input is less than for the small input with our backend. For the small input, there aren't enough iterations to exhaust all cores using the sequentialized versions of some SOACs, and the scheduler will therefore occasionally choose the nested parallel code path, whereas for the medium and large input, it always chooses the sequentialized version. The heuristic determining this choice is briefly mentioned in section 8.

10.6 Parboil

The Parboil benchmark suite contains benchmarks focused on testing performance of "throughput computing architecture and compilers" [SRS+12], and includes applications from several scientific and commercial fields. The reference implementations, later ported to Futhark, are written in OpenCL, OpenMP and CUDA.

For this benchmark suite, we do in general see promising results, except for with the stencil benchmark. For both input sizes of this benchmark, the scheduler chooses sequentialized versions of each SOAC. This leads to yet another example of bad `foreach` placement causing unnecessary scatters and gathers, described in section 12.1. These far outweigh the benefit of vectorizing arithmetic operations, since this benchmark is quite memory-bound.

Settings	Input		Multicore (ms)	Ours (ms)	Speedup
lbm	120_120_150_ldc		7375	6048	x6.20
mri-q	small		125	21	x5.84
	large		649	105	x6.20
sgemm	medium		272	122	x2.23
stencil	small		35	49	x0.72
	default		1531	2065	x0.74

The stencil benchmark demonstrates well how this bad `foreach` placement affects performance by causing gathers and scatters. For example, while stepping through the program, we observed a gather operation using the following memory offsets:

$$\langle 180258, 184354, 188450, 192546, 196642, 200738, 204834, 208930 \rangle$$

With more optimal addressing of memory, which necessitates better placement of `foreach` loops, this gather could have been a vector load. Stencil calculations are fairly common in Futhark's benchmark suite, and the problems described here apply generally to multiple programs that implement them.

10.7 Benchmarking vectorization of mapping sections

We have previously claimed to gain performance by vectorizing the mapping sections of the `redomap` and `scanomap` SOACs, but without much justification. In this section, we aim to empirically validate that some Futhark programs run faster, when we only vectorize the mapping function of a fused SOAC. We also try to justify our approach to vectorization, when the binary operator for the two aforementioned SOACs is a mapped operator. We will use `scanomap` to investigate, but a `redomap` could also have been used, and we have validated that our findings are reproducible when doing so.

10.7.1 Vectorizing mapping function

As a first example, consider the following simple Futhark program using a `scan` and a `map`.

Listing 43: Futhark program with large mapping function

```
def expensive (bound: i64) (x: i64)=
  let y = x * 100 in
  loop (acc: i64) = y for i < bound do
    acc * x

def main bound =
  scan (+) 0 (map (expensive bound) (iota 10000))
```

After fusion, the above Futhark program will only contain a single `scanomap`. Within this SOAC, the mapping function will be `expensive`. We vectorize this part of the `scanomap`, executing the rest sequentially, as described in section 7. For this somewhat contrived program, we do indeed see an increase in performance.

Benchmark		Multicore (ms)	Ours (ms)	Speedup
1000		2	1	x2
10000		13	7	x1.9
100000		127	68	x1.9

Table 13: Benchmark results of multicore backend vs our backend for a program with a non-trivial mapping function.

As shown on the table above, we consistently see a x2 speedup. Using `reduce` instead of `scan`, we are still able to reproduce this speedup.

10.7.2 Vectorizing mapped operators for scan and reduce

As mentioned in section 7, we have different choices for code generation depending on the properties of the binary operator for the SOAC. One of these cases is use of a mapped operator, meaning a perfect `map` nest with an innermost binary operator. Listing 44 shows an example program making use of a mapped operator.

Listing 44: Vectorized scan operator

```
def main [n]  (b: [n]f32) (m: i64): [m][n]f32 =
  let a = replicate n 0.0
  let X = replicate m b
  in scan (map2 (f32.min)) a X
```

The mapped operator in this snippet takes 2 arrays of floats as input and calculates the minimum of them in component-wise fashion. As described in section 7.2.3, instead of vectorizing the `scan` SOAC itself, we will vectorize the mapped operator in this code. We benchmark this program below.

Benchmark		Multicore (ms)	Ours (ms)	Speedup
n=10 m=10000000		73	75	x0.98
n=100 m=1000000		72	63	x1.15
n=10000 m=10000		72	64	x1.13
n=1000000 m=100		215	101	x2.13
n=10000000 m=10		224	73	x3.09

Table 14: Benchmark results of multicore backend vs our backend for a program with a mapped operator.

The above table shows that our approach to vectorizing mapped operators does indeed yield a speedup, and that this speedup increases with the size of the operands passed to the mapped operator, i.e. the input size n.

10.8 Comparison to handwritten ISPC using AOBench

As part of our evaluation, we have ported the AOBench benchmark[18] from ISPC's own benchmark suite to Futhark[19]. This allows us to compare the performance of our backend to the best possible scenario - handwritten, optimized ISPC code. AOBench is a graphical benchmark which calculates ambient occlusion for a simple virtual scene consisting of a few spheres. The version we have ported not only utilizes SIMD by virtue of being written in ISPC, but is also multithreaded. To make the comparisons more fair, we have removed

[18] `https://github.com/ispc/ispc/tree/main/examples/cpu/aobench`
[19] `https://github.com/pema99/aobench-futhark`

the use of a pseudorandom number generator from both the reference implementation, and our Futhark port, and replaced this with a hard-coded sequence of precalculated random numbers. By doing this, we prevent the specific implementation of the chosen pseudorandom number generator from affecting the benchmark results.

Settings		**Reference (ms, speedup)**	**Ours (ms, speedup)**	**Multicore (ms, speedup)**
1024x1024 resolution 2x2 antialiasing 8x8 samples per bounce		77 ms	210 ms x0.37	908 ms x0.08
2048x2048 resolution 2x2 antialiasing 8x8 samples per bounce		292 ms	823 ms x0.35	3502 ms x0.08

Table 15: Results from AOBench benchmark. The first result column shows the reference implementation in handwritten ISPC, the second and third columns show our ISPC backend, and the regular multicore backend. Speedups are calculated from the reference implementation.

Table 15 shows that our backend produces programs which take roughly 3 times longer than the reference implementation to execute. This isn't bad considering how relatively little effort we have put into generating optimized ISPC code. Additionally, the difference between our backend and the unmodified multicore backend is quite large, which makes sense, since AOBench is an almost ideal case for vectorization, being massively parallel in nature, and utilizing mostly simple `map` combinators.

11 Qualitative evaluation

Working with ISPC as a code generation target revealed multiple unforeseen challenges. Therefore, in short paragraphs, we list some of the problems faced, and evaluate the extent to which ISPC is a suitable target language.

11.1 Language shortcomings

First, we list some of the shortcomings we had with ISPC as a language when using it in our code generator. Some issues are related to differences between C and ISPC.

11.1.1 Lack of proper language reference

The documentation provided for the ISPC language is sparse at best. We relied heavily on the User guide [Corb] and Performance guide [Cora] as references when generating ISPC code. However, on several occasions, we discovered language features that were completely missing from any documentation, such as various overloads of atomic operations and cross-lane operations like `extract`. In some situations, we even resorted to reading the source code of the ISPC compiler to determine the expected behavior of a program.

One such case was with the power function, `pow`, which we discovered to not work with negative bases[20] for performance reasons, though some generated programs relied on this. A proper language reference or specification document would have been very helpful while developing the backend. Instead, we had to rely largely on experimentation. This lack of reference was particularly painful when trying to determine the expected behavior of functions from ISPC's standard library. Since speed is often favored over correctness, some functions behave unexpectedly in edge cases, with no indication or warning of this. This is further elaborated in section 11.2.5.

11.1.2 `break` and `return` in varying control flow

We found it to be a strange design decision that `break` and `return` are prohibited within a `foreach`-loop[21], when it *is* allowed within a semantically equivalent version using a `for`-loop with `programCount` and `programIndex`. This quirk alone forced us to implement rather complex error handling, instead of reusing the patterns implemented by Futhark's other backends. In the user guide [Corb] it is explained that a `break` statement in a `for`-loop, with a varying loop index, can be performed by disabling the execution mask before executing the `break` statement. This further begs the question of why a `foreach` loop cannot have `break` or `return`, since the language semantics obviously support it.

11.1.3 Lack of strings

ISPC does not have string literals or string types. As discussed in section 9.6.1, we work around this by calling C code to generate string literals ad-hoc. This is annoying to implement for something that one might reasonably expect from a ”C-based language”. Ignoring the ability to call into C, the lack of strings makes ISPC a less ideal target language, as it complicates error handling.

11.1.4 Vector load/store inference

The ISPC compiler tries to detect at compile time which memory accesses can become vector loads and stores [Cora]. However, its ability to do so seems brittle for large, complex programs. We've tried to rely on this behavior, and were at times affected by this brittle nature. An easy way to explicitly tell the compiler to use a vector operation for a given memory access would be useful.

11.1.5 Variability analysis

Although the ISPC compiler tries to do some inference of variability, its ability to make variables uniform, when it is safe to do so, is not perfect. It sometimes produces surprising results, especially when literals and overloaded functions are involved. Our simple inference algorithm provides better control over when values are treated as uniform, and was necessary to properly generate ISPC code.

[20] `https://github.com/ispc/ispc/issues/1099`

[21] The same applies to `goto`, more info in this issue `https://github.com/ispc/ispc/issues/2279`

11.1.6 Difficulties creating bindings

Calling C functions from ISPC is very easy, provided that all the function arguments are uniform. Whenever some function arguments are varying, we must sequentialize the function calls for each program instance, as described in section 9.3. However, we often found ourselves wanting to call the same function with *either* uniform or varying values, or even a mixture of the two, which we achieved with function overloading. This caused the amount of boilerplate code to grow immensely, though, and was especially tedious when creating bindings for parts of Futhark's C runtime, as these were often written by hand. A simple way to expose a C function to ISPC, and automatically generate overloads for all the valid combinations of variability, would have been very helpful. This could perhaps be achieved by introducing C++ style templated functions, or a polymorphic variability qualifier, which causes several monomorphized overloads to be generated.

11.1.7 Single precision floating point literals

A minor annoyance we encountered, is that floating point literals by default have 32 bit precision [Corb], as opposed to C, which uses 64 bit precision literals. This can be controlled by adding a suffix onto the literal, but annoyingly, this suffix is also incompatible with C. Suffixing `d` onto a literal in ISPC indicates a double precision float, while there is no such suffix for C.

11.2 Bugs

In addition to the aforementioned difficulties, we also encountered a handful of actual bugs in the ISPC compiler.

11.2.1 Inactive lanes affecting program execution

Inactive lanes sometimes affect program execution in strange ways. A notable example of this is the behavior of the division-like operators `div` (division), `mod` (modulo), `quot` (quotient), and `rem` (remainder). These operations do not map to vector instructions, so ISPC is forced to sequentialize them. However, in the interest of speed, ISPC performs the operation for *all* program instances, to avoid the overhead of ensuring that only active instances perform the operation[22]. Since only the results in active lanes are saved, this does not normally cause any issues, but if a run-time error occurs (i.e. a division by zero) on an inactive lane, the program will crash. A workaround for this issue can be seen in listing 45.

Listing 45: Correct varying operation in masked control-flow

```
uint udiv32(uint x, uint y) {
  uint ys = 1; // Safe value for all lanes
  foreach_active(i){
    ys = y; // Active lanes remain the same
  }
  return x / ys; // Operation using the safe values
}
```

[22]https://github.com/ispc/ispc/issues/2169

This type of error was also encountered when indexing an array with a varying index. If an inactive lane had an invalid address, this would in some cases result in a segmentation fault, as the program would try to access memory even for inactive lanes. Strangely, we only encountered this issue when using ISPC's 64-bit addressing mode, which is enabled with a compilation switch.

11.2.2 Order of qualifiers matters for casts

We discovered that the order qualifiers, such as `const`, `uniform` and `varying` matters in typecasts, which isn't the case in C[23]. For example, the typecast `(int uniform * uniform)` causes a compilation error, while `(uniform int * uniform)` does not. Thankfully, the library we use for generation of C (and ISPC) code chooses the latter option automatically.

11.2.3 Struct type can't be used as name

The multicore C backend generates plenty of ad-hoc struct types, which hold all the free variables to be passed to a closure. Presumably to slightly simplify the code generation, the name of each struct type is reused as the variable names for concrete instances of the type, which is completely valid in C. However, ISPC refuses to compile this[24]. Thus, we had to "escape" names of struct-valued variables in quite a few places by adding an underscore to the names.

11.2.4 Internal compiler errors

We encountered quite a few internal compiler errors while generating ISPC code, which would usually produce a completely useless error message[25]. Thankfully, this would mostly occur with ill-formed programs. A very common example of this was when attempting to print a value of a type that can't directly be printed, such as a struct value, for the purpose of debugging.

11.2.5 Incorrect standard library functions

Another place where ISPC favors speed over correctness is with some functions in the standard library, such as `log`. For these functions, ISPC will use some identities to more efficiently calculate the result. Since this is done using bit-level operators, it will behave incorrectly with unusual values, such as infinity and NaN. In table 16, the results of taking the log of positive and negative infinity and a quiet and signalling NaN can be seen.

[23]`https://github.com/ispc/ispc/issues/2281`
[24]`https://github.com/ispc/ispc/issues/2278`
[25]One example: `https://github.com/ispc/ispc/issues/2291`

Function Call	**Result**
log(`INFINITY`)	88.722839
log(`-INFINITY`)	`NaN`
log(`NaN`)	89.128304
log(`-NaN`)	−340.489777

Table 16: Special case results of log function in ISPC

Another example is with the previously mentioned power function, which returns 0 for negative bases.

11.3 Summary

In previous sections, we have discussed some concrete issues and annoyances with ISPC, which we had to overcome to use it as a code generation target. A recurring theme is differences between ISPC and C. ISPC claims to be a ”C-based language” [PM12], but in our experience, the semantics differ enough from C for the claim to be arguable. The language is far from a pure extension of C, and certain features of the latter have been fundamentally changed or removed entirely. Part of the motivation for using ISPC as a code generation target, was that the language was close to C, and we expected to reuse large chunks of code used for C code generation. We did manage this, but weren't able to reuse quite as much as we had hoped.

Outside of being similar to C, the main benefit of using ISPC as a code generation target, is that it lets us generate vectorized code for *multiple* instruction sets and architectures, without having to consider the specifics of these, since the details are abstracted away under fairly simple program semantics. The ISPC language is pretty clearly intended for writing tiny performance-critical subsections of a larger C or C++ codebase, rather than as any kind of substitute for these languages, or as a code generation target. Since the language wasn't designed for the purpose we are using it for, it is no surprise that we have faced some difficulties in doing so. Arguably, something like a lower level version of ISPC (perhaps something closer to one of ISPC's intermediate representations), which provides slightly more control over the generated program, would have been a better suited choice of code generation target. We aren't aware of such a language, though.

With all of this said, the language was robust and sophisticated enough to be successfully used as a tool for programmatically vectorizing generated code for multiple architectures, and we were able to achieve significant speedups on several benchmarks, with only a fairly small amount of changes made to the compilation pipeline. It is undoubtedly easier to extend the multicore backend with vectorization through ISPC, than to rely on generating vector intrinsics directly, and the latter would likely have been a larger undertaking. In summary, we'd describe the language as feasible for our purpose, but not ideal.

Part V

Conclusion

12 Shortcomings and future work

In this section, we describe some shortcomings of our work, including issues we are aware of, but couldn't fit within the scope of the thesis. Where applicable, we describe how these shortcomings might be alleviated in the future.

12.1 Optimizing the compilation pipeline for vectorization

As illustrated on figure 3, we have only made changes to the backend portion of Futhark's compilation pipeline. The goal was primarily to generate ISPC code from the intermediate representation fed to the backend, without much regard for altering it to better enable vectorization. We have implemented a few minor optimizations into the backend, such as use of the AOV representation described in section 10.4, but since these are placed so late in the pipeline, they have limited effect on the generated code. The decision of how to lay out memory would, for example, benefit from being done in a separate pass, earlier in the compilation pipeline.

Another issue we struggled with, and which caused lackluster performance in several benchmarks, was the placement of generated `foreach` loops. We mention in section 8, how code for each SOAC with nested parallelism is generated twice, with one version containing actual nested parallelism, and one version being sequentialized[26]. In the sequentialized version, we vectorize the outermost loop, turning it into a `foreach`. This is the best we can do without changing earlier compilation pipeline stages, but it often hurts performance. Consider the Futhark program below, which multiplies each element of a jagged array with 2.

Listing 46: Nested map SOAC - Futhark

```
def main [n][m] (arr : [n][m]i32) =
  map (map (*2)) arr
```

Given this program, we currently generate roughly the following code for the sequentialized version of the outermost SOAC:

Listing 47: Nested map SOAC - ISPC

```
foreach (i = 0 ... n) {
  for (uniform j = 0; j < m; j++) {
    int v = arr[i*m+j]; // gather!
    arr[i*m+j] = v * 2; // scatter!
  }
}
```

[26]In practice, the sequentialized version of each SOAC is used most of the time.

Notice the non-optimal placement of the foreach. When indexing the array, we end up multiplying a constant onto the varying loop index of the foreach, which causes the array accesses to be expensive gather and scatter operations! Had we instead turned the *innermost* loop into a foreach, both operations would have been efficient vector loads and stores, which is clearly the better choice.

Another option could be to borrow Futhark's *moderate flattening* transformation [Hen17], which is already used for the GPU backends. This technique transforms nested parallelism into flat parallelism by matching nested SOACs against a set of "flattening rules", and changing them accordingly. An example of such a transformation is turning a stack of nested map SOACs into a single loop. Applied to the example in listing 46, we may produce code looking roughly like so:

Listing 48: Flattened map stack - ISPC

```
foreach (i = 0 ... n * m) {
  int v = arr[i];
  arr[i] = v * 2;
}
```

Which is also clearly a more optimal solution than what our backend currently produces. This performance discrepancy will only increase with more levels of nested SOACs. Future work will undoubtedly involve tweaking the compilation pipeline for the ISPC backend, and especially handling the issue of non-optimal foreach placements, likely by building upon the work done for Futhark's GPU pipeline with moderate flattening.

12.2 Further vectorization of scan

As mentioned in the section about data-parallel algorithms for implementing the scanomap SOAC, we never vectorize the scan operation itself - only mapping functions, that have been fused with it. The scan construct is tricky to vectorize, since it has dependencies between loop iterations, but the task is not impossible. [HGLS86] presents a data-parallel algorithm for scans, for which a translation to ISPC is shown in listing 49. In this snippet, binop represents the binary operator used for the scan. Integers are used for simplicity, but the algorithm works with other datatypes as well. The shift intrinsic shifts scalars in a vector register, with the direction of the shift decided by the sign of the input. It works very similarly to a bit shift, but for vector elements.

Listing 49: Data-parallel algorithm for in-register scan

```
int scan_vector(int a) {
  for (uniform int i = 1; i < programCount; i *= 2) {
    a = binop(a, shift(a, -i));
  }
  return a;
}
```

The function presented in the snippet performs a scan over the values in a single vector register, not an arbitrarily sized array. Instead, the function can be used as a building block for a scan algorithm that works on any size of input. Such an algorithm is shown in listing 50. A foreach loop is used to iterate over the array in parallel, and at each iteration, we load a vector of values from the input array, apply the in-register scan shown on listing 49 to it, apply the binary operator to it with the `last` variable as the other input, and write the result back into the array. Then, the `last` variable is set to the last (rightmost) scalar in the vector returned by the in-register scan[27]. The `last` variable is used to handle the dependencies between loop iterations, making sure to carry the final sum calculated by the in-register scan to the next iteration.

Listing 50: Data-parallel algorithm for full scan

```
export void scan_array(uniform int arr[], uniform int size) {
  int last = 0;
  foreach (i = 0 ... size) {
    int val = scan_vector(arr[i]);
    arr[i] = binop(last, val);
    last = broadcast(val, programCount-1);
  }
}
```

The benefit of this algorithm is primarily that the binary operator is only applied $\mathcal{O}(log(n))$ times instead of $\mathcal{O}(n)$, where n is the size of the input array. However, there are some caveats - the algorithm is only applicable if the binary operator used (`binop` in the snippets) can be mapped directly to vector instructions, which is usually the case for many common operations, such as integer addition or boolean disjunction. If the operator cannot be mapped to vector instructions, it must be sequentially applied to each element of the input vector, which negates any performance gain. Additionally, the neutral element for the scan has been assumed to be 0, and the code will malfunction otherwise. An altered example of the in-register scan, which takes other neutral elements into account is shown on listing 51, and uses a masked assignment to prevent the 0's introduced by the `shift` intrinsic from causing issues. This implementation might still perform better than a sequential algorithm for larger vector sizes, despite the additional operations, although we have not verified this.

Listing 51: Data-parallel algorithm for in-register scan with neutral element

```
int scan_vector(int a, int neutral) {
  for (uniform int i = 1; i < programCount; i *= 2) {
    int shifted = shift(a, -i);
    if (shifted == 0) shifted = neutral;
    a = binop(a, shifted);
  }
  return a;
}
```

Future work may include changing the code generation for the `scan` SOAC to utilize this algorithm when it is applicable. Technically, this algorithm is also applicable for reductions, but we neglected to discuss this in detail, since almost every binary operator that can be mapped to vector instructions in the AVX and SSE instruction sets is commutative, meaning we can instead generate a highly efficient interleaved reduction, as described in section 7.2.1.

[27]The `broadcast` intrinsic is almost the same as `extract`, except that it returns a vector with all elements the same, instead of a scalar.

12.3 Parallelizing the compilation pipeline

While our backend robustly generates ISPC code, and passes nearly all of Futhark's tests, it compiles programs quite slowly. Rudimentary profiling of the compilation pipeline reveals that much of this time is actually spent invoking the ISPC compiler. The ISPC compiler, in general, seems to compile large programs quite slowly. This is worsened by the backend being largely single-threaded, and the fact that the size of generated code explodes with the level of nesting of SOACs, since we generate multiple versions of each SOAC. Future work may include parallelizing the backend via multithreading, and running multiple instances of the ISPC compiler simultaneously. Since we currently emit all ISPC kernels into a single, large file, this work would likely involve splitting into multiple, smaller files, which can be handled in parallel. We can make use of the fact that each kernel is completely independent, and perhaps emit each kernel into its own file.

13 Final words

This thesis has presented the design and implementation of a new backend for the purely functional, data-parallel language Futhark, which builds upon the existing multicore C backend, and adds utilization of SIMD by targeting Intel's ISPC language.

We have briefly summarized the history of computer architectures and how they may support parallel computation, and have shown how the parallel nature of modern computer systems may be utilized concretely in the languages relevant to our work, which are primarily ISPC and Futhark.

We have described the central methods upon which this backend builds, including not only the task-parallel algorithms used by the existing multicore backend to compile Futhark's Second-Order Array Combinators, but also additional data-parallel algorithms for these combinators. Furthermore, we describe how these different algorithms may be combined and utilized simultaneously.

We have provided a brief overview of the sections of Futhark's compilation pipeline, which are relevant to our work, and have given extensive descriptions of how the new backend fits into this pipeline, and how we programmatically generate ISPC code. A large chunk of our work was in solving various challenges faced when using ISPC as a target for code generation. As such, we describe how we overcame several of these challenges in order to create a robust code generator, which can correctly handle complex Futhark programs, such as those that may produce error diagnostics.

We have evaluated the new backend both quantitatively and qualitatively. We have run our backend on a large subset of Futhark's benchmark suite, and have compared the performance it achieves against the performance of the existing multicore backend. The benchmarks showed mixed results - our ISPC backend outperforms the existing multicore backend significantly in several benchmarks, but also consistently suffers from slowdowns on more complex programs. We discussed some reasons for these slowdowns, and commented on obvious potential improvements. On top of benchmarks, we have qualitatively evaluated the use of ISPC as a code generation target, and which effect this choice had on the backend. We have commented on some of ISPC's shortcomings, and concluded that, while the language has been sufficient for our use case, it is far from ideal as a code generation target.

Our focus was on creating a robust code generator, rather than a performant backend, so the results of our initial benchmarks are actually quite promising. To conclude the thesis, we describe shortcomings of our work, as well as how our backend may be further developed and improved upon in the future. We believe that, with various improvements that were outside of the scope of our work, the ISPC backend has the potential to be Futhark's most performant CPU-based backend.

References

[ABB$^+$16] Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. Finpar: A parallel financial benchmark. *ACM Trans. Archit. Code Optim.*, 13(2), jun 2016. `https://dl.acm.org/doi/10.1145/2898354`.

[BO15] Randal E Bryant and David R O'Hallaron. *Computer Systems: A Programmer's Perspective, Global Edition*. Pearson Education, London, England, 3 edition, October 2015.

[CBM$^+$09] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009. `https://ieeexplore.ieee.org/document/5306797`.

[Cora] Intel Corporation. Intel® ispc performance guide. `https://ispc.github.io/perfguide.html`.

[Corb] Intel Corporation. Intel® ispc user's guide. `https://ispc.github.io/ispc.html`.

[Cor11] Intel Corporation. Intel® advanced vector extensions programming reference. 2011. `https://www.intel.com/content/dam/develop/external/us/en/documents/36945`.

[Cor12] Intel Corporation. A guide to vectorization with intel® c++ compilers. 2012. `https://www.intel.com/content/dam/www/public/us/en/documents/guides/compiler-auto-vectorization-guide.pdf`.

[Cor16] Intel Corporation. Intel architecture instruction set extensions programming reference. 2016. `https://www.intel.com/content/dam/develop/external/us/en/documents/319433-024-697869.pdf`.

[Ets13] Yoav Etsion. Out-of-order execution, 2013. `https://iis-people.ee.ethz.ch/~gmichi/asocd/addinfo/Out-of-Order_execution.pdf`.

[Hen17] Troels Henriksen. Design and implementation of the futhark programming language (revised). 2017. `https://www.futhark-lang.org/publications/troels-henriksen-phd-thesis.pdf`.

[Hen20] Troels Henriksen. Bounds checking on gpu. 2020. `https://futhark-lang.org/publications/hlpp20.pdf`.

[HGLS86] W. Daniel Hillis and Jr. Guy L. Steele. Data parallel algorithms. *Communications of the ACM, volume 29, issue 12*, 1986. `https://dl.acm.org/doi/10.1145/7902.7903`.

[KSC$^+$12] Changkyu Kim, Nadathur Satish, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey. Closing the ninja performance gap through traditional programming and compiler technology. Technical report, Intel Labs, 2012. `https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-closing-ninja-gap-paper.pdf`.

[MdB19] Dominic Milano Marissa du Bois, Pete Brubaker. Single instruction multiple data made easy with intel® implicit spmd program compiler. 2019. `https://www.intel.com/content/www/us/en/developer/articles/technical/simd-made-easy-with-intel-ispc.html`.

[Oan18] Cosmin E. Oancea. *Lecture Notes for the Software Track of the PMPH Course*. 2018. `https://github.com/diku-dk/pfp-e2020-pub/blob/master/material/flattening/lecture-notes-pmph.pdf`.

[Pha19] Matt Phar. The story of ispc. 2019. `https://pharr.org/matt/blog/2018/04/30/ispc-all`.

[PM12] Matt Pharr and William R. Mark. ispc: A spmd compiler for high-performance cpu programming. 2012. `https://pharr.org/matt/assets/ispc.pdf`.

[SRS$^+$12] John A. Stratton, Cristopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen mei W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing, 2012. `http://impact.crhc.illinois.edu/Shared/Docs/impact-12-01.parboil.pdf`.

[Tra20] Duc Minh Tran. A multicore backend for futhark. 2020. `https://futhark-lang.org/student-projects/duc-msc-thesis.pdf`.

[Wal18] Ingo Wald. Ispc bag of tricks, part 2: On calling back-and-forth between ispc and c/c++. 2018. `https://ingowald.blog/2018/06/25/ispc-bag-of-tricks-part-2-on-the-calling-back-and-forth-between-ispc-and-c-c/`.